2005

Parameter incremental learning algorithm for neural networks

Sheng Wan
West Virginia University

Parameter Incremental Learning Algorithm for Neural Networks

Sheng Wan

Dissertation submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Aerospace Engineering

Larry E. Banta Ph.D., Chair
Nigel N. Clark Ph.D.
Powsiri Klinkhachorn Ph.D.
Roy S. Nutter, Jr. Ph.D.
Gregory J. Thompson Ph.D.

Department of Mechanical and Aerospace Engineering

Morgantown, West Virginia
2005

Keywords: Parameter incremental learning, Neural networks, Incremental learning, Learning algorithms, Backpropagation, Gradient descent, Multi-layer perceptrons, Natural gradient

ABSTRACT

Parameter Incremental Learning Algorithm for Neural Networks

Sheng Wan

In this dissertation, a novel training algorithm for neural networks, named Parameter Incremental Learning (PIL), is proposed, developed, analyzed and numerically validated.

The main idea of the PIL algorithm is based on the essence of incremental supervised learning: that the learning algorithm, i.e., the update law of the network parameters, should not only adapt to the newly presented input-output training pattern, but also preserve the prior results. A general PIL algorithm for feedforward neural networks is accordingly derived, using the first-order approximation technique, with appropriate measures of the performance of preservation and adaptation. The PIL algorithms for the Multi-Layer Perceptron (MLP) are subsequently derived by applying the general PIL algorithm, augmented with the introduction of an extra fictitious input to the neuron. The critical point in obtaining an analytical solution of the PIL algorithm for the MLP is to apply the general PIL algorithm at the neuron level instead of the global network level. The PIL algorithm is basically a stochastic learning algorithm, or on-line learning algorithm, since it adapts the neural weights each time a new training pattern is presented. Extensive numerical study for the newly developed PIL algorithm for MLP is conducted, mainly by comparing the new algorithm with the standard (on-line) Back-Propagation (BP) algorithm. The benchmark problems included in the numerical study are function approximation, classification, dynamic system modeling and neural controller. To further evaluate the performance of the proposed PIL algorithm, comparison with another well-known simplified "high-order" algorithm, i.e., the Stochastic Diagonal Levenberg-Marquardt (SDLM) algorithm, is also conducted.

In all the numerical studies, the new algorithm is shown to be remarkably superior to the standard on-line BP learning algorithm and the SDLM algorithm in terms of 1) the convergence speed, 2) the chance to get rid of the plateau area, which is a frequently encountered problem in standard BP algorithm, and 3) the chance to find a better solution.

Unlike any other advanced or high-order learning algorithms, the PIL algorithm is computationally as simple as the standard on-line BP algorithm. It is also simple to use since, like the standard BP algorithm, only a single parameter, i.e., the learning rate, needs to be tuned. In fact, the PIL algorithm looks just like a "minor modification" of the standard on-line BP algorithm, so it can be applied to any situations where the standard on-line BP algorithm is applicable. It can also replace the standard on-line BP algorithm already in use to get better performance, even without re-tuning of the learning rate.

The PIL algorithm is shown to have the potential to replace the standard BP algorithm and is expected to become yet another standard stochastic (or on-line) learning algorithm for MLP due to its distinguished features.

ACKNOWLEDGEMENTS

I thank my advisor, Dr. Larry E. Banta, for your help, encouragement and support throughout the dissertation process. You helped me in so many ways, particularly during my difficult time, that without your generous support, the completion of my PhD program at WVU would be impossible. I greatly appreciate all that you have done for me.

I also thank Dr. Nigel N. Clark, Dr. Powsiri Klinkhachorn, Dr. Roy S. Nutter, Jr. and Dr. Gregory J. Thompson for serving on my Ph.D. committee and for providing helpful and insightful comments/ suggestions. I would like to thank Dr. Clark for suggesting the use of “data from the real world”. Thanks also to Dr. Thompson for providing me with the engine emission data for testing the new learning algorithm, and for his support and guidance in the area of engine emission modeling.

I deeply appreciate the administration of the Department of Mechanical and Aerospace Engineering, Dr. Ever Barbero, the Chairman, and Dr. Jacky Prucz, the Associate Chairman, for the support not only financially, but also administratively, with their high respect for the future professional career of a PhD candidate.

Finally, I’d like to thank my wife, Yaping, and my son, Ning, for their patience and sacrifices. Without their understanding and support, this would never have happened. I owe them a lot.

TABLE OF CONTENTS

Nomenclature

$g_i(\bullet, \boldsymbol{\theta}_i)$:	$\Re^{n_i} \to \Re$ a smooth function denoting the i^{th} neuron of a neural network with $\boldsymbol{\theta}_i$ being its adjustable parameter vector, $i = 1, 2, \cdots, L$
$\mathbf{I}$:	Identity matrix with appropriate dimension
L:	the total number of neurons in the neural network
$m(\bullet, \bullet)$:	a distance metric between two functions
$\Re^n$:	Euclidian space with dimension *n*
$\|\mathbf{x}\|$:	$= \left(\mathbf{x}^{\mathbf{T}}\mathbf{x}\right)^{1/2}$, the Euclidean norm of a vector
$\|\mathbf{x}\|_{\mathbf{M}}$:	$= \left(\mathbf{x}^{\mathbf{T}}\mathbf{M}\mathbf{x}\right)^{1/2}$, $\mathbf{M} > 0$, the weighted Euclidean norm of a vector
$\boldsymbol{\Psi}(\bullet, \boldsymbol{\theta})$:	$\Re^{\ell} \to \Re^m$, a smooth function representing the map of a neural network, or an *approximator*, with adjustable parameter $\boldsymbol{\theta} \in \Re^M$
$\cong$:	two quantities are approximately equal in the sense of first-order approximation
BP	Back-Propagation
GRD	Gradient Descent
NGD	Natural Gradient Descent
PIL	Parameter Incremental Learning

A quantity $\mathbf{F}(\mathbf{x})$, which depends on another quantity $\mathbf{x}$, will sometimes be abbreviated as $\mathbf{F}$, when it can be done without causing confusion.

List of figures

List of tables

Chapter 1 Introduction

The problem of learning from input-output data, or the so-called *supervised learning*, from the view point of function approximation, can be cast as a problem of using a known function (or so-called *approximator*) with a set of adjustable parameters to approximate an (unknown) *underlying function* through the observed data. To the best knowledge of the author, all the (parameterized) supervised learning algorithms in the literature are derived from some sort of optimization problems over the adjustable parameter set, where the objective is to optimize a certain cost function (or *error function*) which measures the discrepancy between the observed output and the approximated output with given input. The most frequently used technique in searching solutions is the gradient-descent algorithm, particularly the stochastic (or on-line) gradient-descent algorithm. The gradient-descent learning algorithm for neural networks, and many of its variants, have been extensively investigated and widely practiced, see e.g. [1]-[4] and the references therein. It is worth noting that although many of the variants of the gradient-descent algorithm and other "advanced algorithms" have been proposed, the basic gradient-descent algorithm, though often criticized as slow in convergence, still remains most popular for its simplicity, efficiency and robustness.

In a gradient-descent learning algorithm, the prediction error of the neural network is reduced after an iteration of applying the learning algorithm over the cost function. It is noted, however, that no specific measure is explicitly taken to preserve the "already learned" property of the approximator during the learning iteration. On the other hand, in view of the learning approaches of human beings, it seems natural to build posterior learning results upon prior results. This is essentially a feature of *incremental learning*: it basically means that after the approximator makes an adaptation to the new input-output pattern by adjusting its parameters, two goals (though somewhat contradictive) are simultaneously achieved: First, the prediction error of the neural network is reduced (if not completely eliminated) for this new pattern; Second, the perturbation of the approximator (measured by some well-defined metric) caused by the parameter adaptation is kept as small as possible, thus the *a priori* result is best preserved. It appears that most of the known

learning algorithms such as gradient-descent only address the first goal, *i.e.*, adaptation to the training data, without explicitly addressing the issue of preservation of *a priori* results. In this dissertation, a supervised learning approach is proposed which simultaneously addresses the two goals in a unified framework. It is noted that unlike some other incremental learning methods in neural networks such as Resource Allocation Networks proposed in [5] for Radial Basis Networks, where the spirit of incremental learning is achieved mainly by structural adaptation (*i.e.*, it recruits a new neuron, or allocates new *resources*, whenever necessary), this research concentrates on the situation where only parameter adaptation is considered. Thus the proposed method is called the *Parameter Incremental Learning* (PIL) strategy.

Chapter 2 Brief review of the learning algorithms for feed-forward neural networks

This chapter briefly reviews different methods for training feedforward neural networks. Feedforward neural networks are widely used to solve complex problems in function approximation, pattern classification, system modeling and identification, time series analysis, signal processing, and control systems [6]. The theoretical foundation of feedforward neural networks in diverse applications is that feedforward neural networks are capable of approximating any reasonably smooth functions, provided a sufficient number of neurons are used. In other words, feedforward neural networks are *universal approximators* [7], [8]. Typical feedforward neural networks include Multi-Layer Perceptron (MLP) and Radial Basis Function Networks (RBFN).

An important property of the feedforward neural networks is their ability to *learn* from the input-output data (*supervised learning*). Let $(\mathbf{x_1}, \mathbf{y_1})$, $(\mathbf{x}_2, \mathbf{y}_2)$, …, $(\mathbf{x_N}, \mathbf{y_N})$ (the $\mathbf{x}_i$, $\mathbf{y}_i$ ($i = 1, 2, \cdots, N$) are the inputs and outputs of the *underlying system* that generates the data set, respectively) be the set of input-output data (or *patterns*) used for neural network training. The task of neural network training is to find a set of adjustable parameters $\boldsymbol{\theta}$ of the neural network $\boldsymbol{\Psi}$ such that the input-output map of the neural network

$$\hat{\mathbf{y}} = \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta}) \tag{1}$$

approximates (in some sense) the *underlying function* reflected in the given training data set of input-output pairs. That is, the difference between the desired target values $\mathbf{y}_i$ and the approximated values

$$\hat{\mathbf{y}}_i = \boldsymbol{\Psi}(\mathbf{x}_i, \boldsymbol{\theta}) \tag{2}$$

given by the neural network for a given input data $\mathbf{x}_i$ in the training set, should be made as small as possible for all $i = 1, 2, \cdots, N$. Normally, an *error function* or *cost function* can be defined to measure the agreement between the desired output of the training data and the approximated output. Let $E(\boldsymbol{\theta})$ denote the error function or cost function that measures learning performance, *i.e.*, the discrepancy between the desired output

and the approximated output. Most often, the $E(\boldsymbol{\theta})$ is selected as the Mean Square Error (MSE), defined as:

$$E(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^{\mathbf{T}}(\mathbf{y}_i - \hat{\mathbf{y}}_i) \tag{3}$$

Most of the learning algorithms for neural networks are derived from some sort of optimization problems aimed at minimizing the cost function $E(\boldsymbol{\theta})$. The most popular optimization algorithm is the basic gradient descent algorithm. The gradient descent algorithm is based on a linear (or first-order) approximation of the error function given by

$$E(\boldsymbol{\theta} + \delta\boldsymbol{\theta}) \cong E(\boldsymbol{\theta}) + \delta\boldsymbol{\theta}^{\mathbf{T}} \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{4}$$

The parameter update (*i.e*., the learning algorithm) is

$$\delta\boldsymbol{\theta} = -\eta \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}, \quad \eta > 0 \tag{5}$$

The step size or learning rate η can be determined by a line search (corresponding to the so-called *steepest gradient descent*) but is usually set to a small constant in most of the applications. Repeated application of the above parameter update until the $E(\boldsymbol{\theta})$ becomes small enough constitutes the standard gradient descent (batch) learning algorithm.

An alternative version of the above standard gradient descent, which is perhaps more popular in applications, is the *on-line* (or *stochastic*) gradient descent learning algorithm. In the on-line gradient descent algorithm, the parameters of the neural network are updated based only on a single training pattern. That is, the parameter update is the same as the one defined by Eq. (5) except that the error function is defined with respect to only a single pattern. In the on-line gradient descent algorithm, a full presentation of all patterns in the training set is called an *epoch*. The on-line method is often preferred particularly when training data set is large and contains redundant information. In addition, on-line method is indispensable when the underlying

function being modeled is changing over time, a quite common scenario in control or time series prediction where the system changes gradually over time.

For MLP, the gradient descent algorithm is also known as the error Back-Propagation (BP) method [9],[10],[11],[12], since in calculating the gradient, *i.e.*, the partial derivatives of the error function $E(\boldsymbol{\theta})$ with respect to the parameter $\boldsymbol{\theta}$, the BP strategy, which is essentially the chain-rule in calculating the derivatives, is employed.

Since the "rediscovery" of the BP method, the standard gradient descent has been widely practiced, and continues to be a conventional, and perhaps the most popular algorithm in many applications due to its simplicity, efficiency and robustness. It also has become a "benchmark" neural network training algorithm for studying other neural network training algorithms. However, the standard gradient descent algorithm has often been criticized as extremely slow in convergence in many applications. The slow convergence is mainly caused by the *plateau phenomenon* which can dominate the whole learning process in the standard gradient descent learning algorithm [13].

Lots of efforts have been made to improve the convergence speed of the standard gradient descent learning algorithm. Those following are among the best known methods, other than the standard gradient descent algorithms, for training feedforward neural networks.

1) Adding a momentum term

Quite often, the learning process can experience some oscillation phenomena, which consequently leads to slow convergence of the learning. A common remedy to damp out the oscillation is to add an extra term in the standard gradient descent algorithm [14]:

$$\delta\boldsymbol{\theta}_{k+1} = -\eta \frac{\partial E(\boldsymbol{\theta}_k)}{\partial \boldsymbol{\theta}_k} + \alpha \delta\boldsymbol{\theta}_k, \quad \eta > 0, \alpha > 0 \tag{6}$$

The extra term $\alpha\delta\boldsymbol{\theta}_k$ is called *momentum*, which, by taking the past gradient descent into account, has the

effect of smoothing the direction of gradient descent to prevent oscillation on the surface of a non-linear function $E(\boldsymbol{\theta})$ that forms a curving canyon. Although this technique works well for some applications, it can sometimes result in even worse convergence speed [3].

2) Adaptive learning rate

A popular strategy in accelerating the convergence speed of the conventional gradient descent algorithm is to use a variable learning rate (so-called *adaptive learning*), rather than a constant one in the standard gradient descent algorithm. Many adaptive learning rate methods, mostly heuristic, are proposed, mainly by increasing or decreasing the learning rate based on information from the error or error rate. Quickprop [15] (which is a method somewhere between the gradient descent method and the Newton method) and Resilient Propagation [16] (Rprop) are typical adaptive learning methods. It is noted that adaptive learning approaches can only be applied in batch learning.

3) Second-order optimization methods (Newton's method and its variations)

The essence of the second-order approach in neural network learning is to accelerate the convergence speed of the learning algorithm by making use of second-order, or *curvature* information about the error function $E(\boldsymbol{\theta})$. The following second-order approximation of the cost function can be obtained by using the first three terms of the Taylor-series expansion of the current network parameter:

$$E(\boldsymbol{\theta}+\delta\boldsymbol{\theta}) \approx E(\boldsymbol{\theta}) + \delta\boldsymbol{\theta}^{\mathbf{T}} \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} + \frac{1}{2}\delta\boldsymbol{\theta}^{\mathbf{T}} \frac{\partial^2 E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2}\delta\boldsymbol{\theta} \tag{7}$$

The "optimal" $\delta\boldsymbol{\theta}$, known as the Newton's step, is given by:

$$\delta\boldsymbol{\theta} = -\left(\frac{\partial^2 E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2}\right)^{-1} \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{8}$$

which will bring the right side of Eq. (7) to minimum in just one step (if the Hessian $\frac{\partial^2 E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2}$ is positive

definite). This method is called *Newton's method*. However, this "pure" Newton's method can rarely be used in applications, because in practice the Hessian is usually ill-conditioned (singular or near-singular), and can rarely be positive definite, and that will cause the algorithm to blow up. To deal with these difficulties, several *modified Newton's methods* were proposed, mostly resulting from approximating the Hessian by various methods such that the approximated Hessian is guaranteed to be invertible and positive-definite. The Gauss-Newton method and the Levenberg-Marquardt method are the typical modified Newton's methods.

It is noted that the second-order methods are usually restricted to those applications where the dimension of the parameter space is small, due to the computation burden associated with the inversion of a matrix whose dimension equals the number of free parameters in the neural network. Second-order methods are mainly designed for batch learning.

4) Conjugate gradient [17], [1]

The conjugate gradient method is one of the most popular methods in nonlinear optimization. It bears a mixture flavor of the gradient descent and the Newton's method. Basically, the conjugate gradient method attempts to find descent directions that try to minimally spoil the result achieved in the previous iterations, while avoiding the requirement to manipulate the Hessian matrix. It uses a line search in the procedure. One drawback of the conjugate gradient method is that it works only for batch learning.

5) Natural gradient descent method

Natural gradient descent is a recently proposed novel and theoretically elegant method [18]. This method is based on the information geometry and uses the Riemann metric of the parameter space to define the steepest descent direction of a cost function. It is basically a kind of stochastic on-line learning method. The learning algorithm is expressed as

$$\delta\boldsymbol{\theta} = -\eta G^{-1}\left(\boldsymbol{\theta}\right)\frac{\partial E\left(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}\right)}{\partial\boldsymbol{\theta}}, \quad \eta > 0 \tag{9}$$

where the matrix $G\left(\boldsymbol{\theta}\right)$ is the Riemann metric tensor. For a statistical model of stochastic learning, the

Riemann metric tensor is given by the Fisher information matrix. The natural gradient descent is shown to be superior to the conventional gradient descent in that it has a much better chance of avoiding the plateau area in the learning process [19]. However, the Fisher information matrix of multi-layer perceptrons can hardly ever be obtained, and even if it is available, its inversion is computationally expensive.

Chapter 3 Parameter Incremental Learning Problem

3.1 Parameter Incremental Learning Problem at global level

Parameter incremental learning belongs to the category of on-line learning since the parameter updates take place whenever a new pattern is available for training. Basically, the parameter incremental learning strategy is a trade off between the adaptation to the new input-output pattern and the preservation of the *a priori* result of the neural network when the learning algorithm responds to the new input-output patterns. In this regard, two performance indices need to be simultaneously addressed: the performance of adaptation and the performance of preservation. For a given input-output pair of training data $(\mathbf{x}, \mathbf{y})$, the (weighted) squared error between the target value and the predicted output by the neural network is chosen to measure the performance of adaptation, expressed as

$$J^{adpt} = \frac{1}{2}\Delta\mathbf{y}_{\mathbf{new}}^{\mathbf{T}}\Xi\Delta\mathbf{y}_{\mathbf{new}} \tag{10}$$

where the matrix Ξ is a positive-definite weight matrix, and

$$\Delta\mathbf{y}_{\mathbf{new}} = \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta} + \delta\boldsymbol{\theta}) - \mathbf{y} \tag{11}$$

is the *post-prediction error*, *i.e.*, the prediction error of the neural network after the parameter is updated from $\boldsymbol{\theta}$ to $\boldsymbol{\theta} + \delta\boldsymbol{\theta}$. On the other hand, the performance index measuring the neural network's deformation caused by the parameter adaptation is chosen to be the commonly used squared L^2-norm induced metric of the distance between functions, *i.e.*,

$$J^{pres} = \frac{1}{2}m_{L^2}^2(\boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta} + \delta\boldsymbol{\theta}), \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})) \tag{12}$$

where

$$m_{L^2}(\boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta} + \delta\boldsymbol{\theta}), \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})) = \left(\int_{\mathbf{x}\in\mathfrak{D}} \left\|\boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta} + \delta\boldsymbol{\theta}) - \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})\right\|^2 d\mathbf{x}\right)^{1/2} \tag{13}$$

and $\mathfrak{D}$ is a properly defined region of the input to the network. Ideally, we want both J^{adpt} and J^{pres} be as small as possible after the parameter update. Thus, the cost function that captures the essence of the PIL, which is basically a tradeoff between the two (somewhat contradictive) performance indices, is obtained by combining the two performance indices (10) and (12) as the following single cost function:

$$J = J^{adpt} + \lambda J^{pres} \tag{14}$$

where $\lambda > 0$ is a user-selected design parameter.

Thus, the following general PIL Problem (at the global level) is posed:

General PIL Problem at Global Level: Given a pair of input-output training vectors $(\mathbf{x}, \mathbf{y})$, find the parameter increment $\delta\boldsymbol{\theta}$ to the modifiable parameter $\boldsymbol{\theta}$ of the neural network $\boldsymbol{\Psi}$ such that the cost function

$$J = \frac{1}{2}\Delta\mathbf{y}_{\mathbf{new}}^{\mathbf{T}}\Xi\Delta\mathbf{y}_{\mathbf{new}} + \lambda\frac{1}{2}\int_{\mathbf{x}\in\mathfrak{D}}\left\|\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta}+\delta\boldsymbol{\theta}) - \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})\right\|^2 d\mathbf{x} \tag{15}$$

is minimized.

Generally, an analytical solution for this nonlinear optimization problem is difficult to obtain. However, this sophisticated non-linear optimization problem can be easily "solved" when using the first-order approximation of the neural network with respect to the (small) parameter perturbation, as is given by the following theorem.

Theorem 1 (General PIL Algorithm at global level) The first-order approximate solution to the general PIL problem is given by

$$\delta\boldsymbol{\theta}^{\mathbf{PIL}} \triangleq -\frac{1}{\lambda}\boldsymbol{\Theta}^{-1}(\boldsymbol{\theta})\frac{\partial\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\Omega\Delta\mathbf{y} \tag{16}$$

where

$$\Delta\mathbf{y} \triangleq \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta}) - \mathbf{y} \tag{17}$$

is the *a priori-prediction error, i.e.*, the prediction error of the neural network prior to parameter updating,

and

$$\boldsymbol{\Theta}(\boldsymbol{\theta}) \triangleq \int_{\mathbf{x}\in\mathfrak{D}} \frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} d\mathbf{x} \tag{18}$$

$$\Omega \triangleq \left(\Xi^{-1} + \frac{1}{\lambda} \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \boldsymbol{\Theta}^{-1}(\boldsymbol{\theta}) \frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{-1} \tag{19}$$

Proof:

Note that by using the first-order approximation of the neural network with respect to the parameter perturbation, we have

$$\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta}+\delta\boldsymbol{\theta}) \cong \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta}) + \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \delta\boldsymbol{\theta} \tag{20}$$

Then

$$\begin{aligned} &\int_{\mathbf{x}\in\mathfrak{D}} \left\| \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta}+\delta\boldsymbol{\theta}) - \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta}) \right\|^2 \cong \int_{\mathbf{x}\in\mathfrak{D}} \left(\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \delta\boldsymbol{\theta} \right)^{\mathbf{T}} \left(\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \delta\boldsymbol{\theta} \right) d\mathbf{x} \\ &= \delta\boldsymbol{\theta}^{\mathbf{T}} \left(\int_{\mathbf{x}\in\mathfrak{D}} \frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} d\mathbf{x} \right) \delta\boldsymbol{\theta} = \delta\boldsymbol{\theta}^{\mathbf{T}} \boldsymbol{\Theta}(\boldsymbol{\theta}) \delta\boldsymbol{\theta} \end{aligned} \tag{21}$$

On the other hand,

$$\begin{aligned} \Delta\mathbf{y}_{\mathbf{new}} &= \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta}+\delta\boldsymbol{\theta}) - \mathbf{y} \\ &= \left(\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta}+\delta\boldsymbol{\theta}) - \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta}) \right) + \left(\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta}) - \mathbf{y} \right) \\ &\cong \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \delta\boldsymbol{\theta} + \Delta\mathbf{y} \end{aligned} \tag{22}$$

Therefore the cost function J given by Eq. (15) can be approximated as

$$J \cong \frac{1}{2} \left(\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \delta\boldsymbol{\theta} + \Delta\mathbf{y} \right)^{\mathbf{T}} \Xi \left(\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \delta\boldsymbol{\theta} + \Delta\mathbf{y} \right) + \frac{\lambda}{2} \delta\boldsymbol{\theta}^{\mathbf{T}} \boldsymbol{\Theta}(\boldsymbol{\theta}) \delta\boldsymbol{\theta} \tag{23}$$

The optimal solution to the *quadratic optimization problem* with the cost function given by Eq. (23) can be obtained from the following optimality condition:

$$\frac{\partial J}{\partial \delta \boldsymbol{\theta}} \cong \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \Xi \left(\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \delta \boldsymbol{\theta} + \Delta \mathbf{y} \right) + \lambda \boldsymbol{\Theta}(\boldsymbol{\theta}) \delta \boldsymbol{\theta} \triangleq 0 \tag{24}$$

The solution to the Eq. (24) is given by:

$$\begin{aligned} \delta \boldsymbol{\theta}^{\mathbf{PIL}} &= -\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \Xi \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} + \lambda \boldsymbol{\Theta}(\boldsymbol{\theta}) \right)^{-1} \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \Xi \Delta \mathbf{y} \\ &= -\frac{1}{\lambda} \boldsymbol{\Theta}^{-1}(\boldsymbol{\theta}) \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \left(\Xi^{-1} + \frac{1}{\lambda} \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \boldsymbol{\Theta}^{-1}(\boldsymbol{\theta}) \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{-1} \Delta \mathbf{y} \\ &= -\frac{1}{\lambda} \boldsymbol{\Theta}^{-1}(\boldsymbol{\theta}) \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \Omega \Delta \mathbf{y} \end{aligned} \tag{25}$$

This completes the proof.

Apparently, for a specific type of feedforward neural network, the key to finding an (approximate) analytical solution amounts to finding an analytical formula to the integral (18) (and preferably its inverse for the sake of reducing the computational burden). It is noted that, unfortunately, it is generally extremely difficult (if not impossible) to find such a solution for general (nonlinear) neural networks. To overcome the difficulty, a sub-optimal strategy will be proposed in the next section of this chapter.

3.2 Parameter Incremental Learning Problem at neuron level

Note that the purpose here is to obtain a learning algorithm by deriving an (approximate) analytical solution to the optimization problem. The General PIL Algorithm at the global level, as was argued in the previous section, is not, in general, an analytical solution. A sub-optimal strategy is proposed in this section, aiming at obtaining an analytical solution while maintaining the spirit of the parameter incremental learning. The key idea of the strategy is to define an index which measures the neural network preservation

performance at the neuron level, instead of at the "global level" as is defined by Eq. (12). This implies that the performance index is measured against the deformation of the individual neurons instead of against the entire neural network as a whole. A suitable performance index, which will simplify the problem while maintaining the essence of the preservation feature of the PIL, is:

$$\tilde{J}^{pres} = \frac{1}{2}\sum_{i=1}^{L} \lambda_i m_{L^2}^2 \left(g_i(\mathbf{u}_i, \boldsymbol{\theta}_i + \delta\boldsymbol{\theta}_i),\, g_i(\mathbf{u}_i, \boldsymbol{\theta}_i) \right) \tag{26}$$

where the distance metric

$$m_{L^2} \left(g_i(\mathbf{u}_i, \boldsymbol{\theta}_i + \delta\boldsymbol{\theta}_i),\, g_i(\mathbf{u}_i, \boldsymbol{\theta}_i) \right) = \left(\int_{\mathbf{u}_i \in \mathfrak{D}_i} \left(g_i(\mathbf{u}_i, \boldsymbol{\theta}_i + \delta\boldsymbol{\theta}_i) - g_i(\mathbf{u}_i, \boldsymbol{\theta}_i) \right)^2 d\mathbf{u}_i \right)^{\frac{1}{2}}, \quad i = 1, 2, \cdots, L \tag{27}$$

is a measure of the "deformation" of a single neuron caused by its own parameter perturbation. λ_i ($i = 1, \cdots, L$) are positive numbers (to be designed). The variable vector $\mathbf{u}_i$ is the input to the i^{th} neuron $\mathbf{g}_i$. It is assumed that $\left(g_i(\mathbf{u}_i, \boldsymbol{\theta}_i + \delta\boldsymbol{\theta}_i) - g_i(\mathbf{u}_i, \boldsymbol{\theta}_i) \right)^2$ is integrable over a properly defined region $\mathfrak{D}_i$ for all $i = 1, \cdots, L$. It will be seen in Chapter 5 that with the proposed performance index given by Eq. (26) and Eq. (27), the PIL algorithm for MLP neural network is solvable (with the aid of some mathematical tricks).

Thus, the following general PIL Problem (at the neuron level) is posed:

General PIL Problem at Neuron Level: Given a pair of input-output training data vectors $(\mathbf{x}, \mathbf{y})$, find the parameter increment $\delta\boldsymbol{\theta}$ to the modifiable parameter $\boldsymbol{\theta}$ of the neural network $\boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})$ such that the following cost function is minimized:

$$\tilde{J} = \frac{1}{2}\boldsymbol{\Delta}\mathbf{y}_{\mathbf{new}}^{\mathbf{T}} \boldsymbol{\Xi} \boldsymbol{\Delta}\mathbf{y}_{\mathbf{new}} + \frac{1}{2}\sum_{i=1}^{L} \lambda_i \int_{\mathbf{u}_i \in \mathfrak{D}_i} \left(g_i(\mathbf{u}_i, \boldsymbol{\theta}_i + \delta\boldsymbol{\theta}_i) - g_i(\mathbf{u}_i, \boldsymbol{\theta}_i) \right)^2 d\mathbf{u}_i \tag{28}$$

Similar to the General PIL Problem at Global Level in the previous section**,** the solution is given by following theorem.

Theorem 2 (General PIL Algorithm at Neuron Level) The first-order approximate solution to the general PIL problem at neuron level is given by

$$\delta\boldsymbol{\theta}_i^{\mathbf{PIL}} \triangleq -\frac{1}{\lambda_i}\boldsymbol{\Theta}_i^{-1}(\boldsymbol{\theta}_i)\frac{\partial\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial\boldsymbol{\theta}_i}\Omega\Delta\mathbf{y}, \qquad i=1,\cdots,L \tag{29}$$

where

$$\boldsymbol{\Theta}_i(\boldsymbol{\theta}_i) \triangleq \int_{\mathbf{u}_i\in\mathfrak{D}_i} \frac{\partial g_i(\mathbf{u}_i,\boldsymbol{\theta}_i)}{\partial\boldsymbol{\theta}_i}\left(\frac{\partial g_i(\mathbf{u}_i,\boldsymbol{\theta}_i)}{\partial\boldsymbol{\theta}_i}\right)^{\mathbf{T}} d\mathbf{u}_i, \qquad i=1,\cdots,L \tag{30}$$

$$\Omega \triangleq \left(\Xi^{-1}+\left(\frac{\partial\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right)^{\mathbf{T}}\boldsymbol{\Theta}_\lambda^{-1}(\boldsymbol{\theta})\frac{\partial\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right)^{-1} \tag{31}$$

assuming that each $\boldsymbol{\Theta}_i(\boldsymbol{\theta}_i)$ is invertible. And

$$\boldsymbol{\Theta}_\lambda(\boldsymbol{\theta}) \triangleq diag\left[\lambda_1\boldsymbol{\Theta}_1 \quad \lambda_2\boldsymbol{\Theta}_2 \quad \cdots \quad \lambda_L\boldsymbol{\Theta}_L\right] \tag{32}$$

Proof:

By using the first-order approximation of the neuron with respect to the parameter perturbation, we have

$$g_i(\mathbf{u}_i,\boldsymbol{\theta}_i+\delta\boldsymbol{\theta}_i) \cong g_i(\mathbf{u}_i,\boldsymbol{\theta}_i)+\left(\frac{\partial g_i(\mathbf{u}_i,\boldsymbol{\theta}_i)}{\partial\boldsymbol{\theta}_i}\right)^{\mathbf{T}}\delta\boldsymbol{\theta}_i, \qquad i=1,\cdots,L \tag{33}$$

Thus the equation (27) is approximated as

$$\begin{aligned} m_{L^2}^2\left(g_i(\mathbf{u}_i,\boldsymbol{\theta}_i+\delta\boldsymbol{\theta}_i),\, g_i(\mathbf{u}_i,\boldsymbol{\theta}_i)\right) &\cong \int_{\mathbf{u}_i\in\mathfrak{D}_i}\left(\left(\frac{\partial g_i(\mathbf{u}_i,\boldsymbol{\theta}_i)}{\partial\boldsymbol{\theta}_i}\right)^{\mathbf{T}}\delta\boldsymbol{\theta}_i\right)^{\mathbf{T}}\left(\left(\frac{\partial g_i(\mathbf{u}_i,\boldsymbol{\theta}_i)}{\partial\boldsymbol{\theta}_i}\right)^{\mathbf{T}}\delta\boldsymbol{\theta}_i\right)d\mathbf{u}_i \\ &= \delta\boldsymbol{\theta}_i^{\mathbf{T}}\left(\int_{\mathbf{u}_i\in\mathfrak{D}_i}\frac{\partial g_i(\mathbf{u}_i,\boldsymbol{\theta}_i)}{\partial\boldsymbol{\theta}_i}\left(\frac{\partial g_i(\mathbf{u}_i,\boldsymbol{\theta}_i)}{\partial\boldsymbol{\theta}_i}\right)^{\mathbf{T}}d\mathbf{u}_i\right)\delta\boldsymbol{\theta}_i = \delta\boldsymbol{\theta}_i^{\mathbf{T}}\boldsymbol{\Theta}_i(\boldsymbol{\theta}_i)\delta\boldsymbol{\theta}_i, \qquad i=1,\cdots,L \end{aligned} \tag{34}$$

Also note that the adjustable parameter vector $\boldsymbol{\theta}$ to the entire neural network can be written as the concatenation of all the parameter vectors of each of the neurons in the neural network, *i.e.*,

$$\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\theta}_1^{\mathbf{T}} & \boldsymbol{\theta}_2^{\mathbf{T}} & \cdots & \boldsymbol{\theta}_L^{\mathbf{T}} \end{bmatrix}^{\mathbf{T}} \tag{35}$$

Then the index of preservation performance (26) can be approximated as

$$\tilde{J}^{\,pres} \cong \frac{1}{2}\sum_{i=1}^{L} \lambda_i \delta\boldsymbol{\theta}_i^{\mathbf{T}} \boldsymbol{\Theta}_i(\boldsymbol{\theta}_i)\delta\boldsymbol{\theta}_i = \frac{1}{2}\delta\boldsymbol{\theta}^{\mathbf{T}} \boldsymbol{\Theta}_\lambda(\boldsymbol{\theta})\delta\boldsymbol{\theta} \tag{36}$$

Therefore by using Eq. (22) and Eq. (36), the cost function (28) can be approximated as

$$\tilde{J} \cong \frac{1}{2}\left(\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{\mathbf{T}} \delta\boldsymbol{\theta} + \Delta\mathbf{y}\right)^{\mathbf{T}} \Xi \left(\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{\mathbf{T}} \delta\boldsymbol{\theta} + \Delta\mathbf{y}\right) + \frac{1}{2}\delta\boldsymbol{\theta}^{\mathbf{T}} \boldsymbol{\Theta}_\lambda(\boldsymbol{\theta})\delta\boldsymbol{\theta} \tag{37}$$

The optimal solution to this quadratic optimization problem with the cost function given by Eq. (37) can be obtained from the following optimality condition:

$$\frac{\partial \tilde{J}}{\partial \delta\boldsymbol{\theta}} \cong \frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \Xi \left(\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{\mathbf{T}} \delta\boldsymbol{\theta} + \Delta\mathbf{y}\right) + \boldsymbol{\Theta}_\lambda(\boldsymbol{\theta})\delta\boldsymbol{\theta} \triangleq 0 \tag{38}$$

The solution to (38) is given by:

$$\begin{aligned} \delta\boldsymbol{\theta}^{\mathbf{PIL}} &= -\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \Xi \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{\mathbf{T}} + \boldsymbol{\Theta}_\lambda(\boldsymbol{\theta})\right)^{-1} \frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \Xi \Delta\mathbf{y} \\ &= -\boldsymbol{\Theta}_\lambda^{-1}(\boldsymbol{\theta}) \frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \left(\Xi^{-1} + \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{\mathbf{T}} \boldsymbol{\Theta}_\lambda^{-1}(\boldsymbol{\theta}) \frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{-1} \Delta\mathbf{y} \\ &= -\boldsymbol{\Theta}_\lambda^{-1}(\boldsymbol{\theta}) \frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \Omega \Delta\mathbf{y} \end{aligned} \tag{39}$$

and this is equivalent to (29). This completes the proof.

It seems, when comparing the equation (39) (or equivalently, (29)) with the equation (16), that the main difference between the algorithms of the PIL at two different levels, *i.e.*, the global level and the neuron level, is that at the neuron level, the square matrix $\boldsymbol{\Theta}_\lambda(\boldsymbol{\theta})$ given by (32) has a block-diagonal structure with the dimension of each of the blocks being equal to the number of parameters of the corresponding neuron. It

is noted, however, that the PIL algorithm at the neuron level has much more important meaning than that. As will be seen in Chapter 5, that the integrals, defined by Eq. (30), have an analytical solution for the Multi-Layer Perceptrons.

3.3 An alternative perspective for the PIL Problem at neuron level

To gain further insights for the PIL algorithm at neuron level presented in previous section, it is of interest to take a look at the problem from an alternative perspective. In this section, the PIL problem at neuron level will be posed from a different perspective yet is able to capture the same essence of incremental learning (at the neuron level) as previously discussed. The main point here is to pose the problem as if only a single neuron in the network were to be adapted when a training pattern is presented, and the performance index which bears the essence of incremental learning is posed so as to minimize the deformation of this single neuron, subject to a certain amount of error reduction (i.e., adaptation to the training pattern) specified by an equality constraint. Quite interestingly, it will be shown in what follows that, though started from a different perspective, the resulting algorithm is essentially the same as the one obtained in the previous section.

To simplify the presentation, it is assumed that the neural network has a single output. The alternative version of the PIL problem at the neuron level is posed as follows.

Alternative version of the General PIL Problem at the neuron level: Given a new pair of input-output training data $(\mathbf{x}, \mathbf{y})$, find the increment of the parameter vector $\boldsymbol{\theta}_i$ of the i^{th} neuron in the neural network, $\delta\boldsymbol{\theta}_i$, that minimizes the cost function, or the deformation of the neuron caused by the variation of the parameter vector:

$$J_i\left(\boldsymbol{\theta}_i, \delta\boldsymbol{\theta}_i\right) = \frac{1}{2}\int_{\mathbf{u}_i \in \mathfrak{D}_i} \left(g_i(\mathbf{u}_i, \boldsymbol{\theta}_i + \delta\boldsymbol{\theta}_i) - g_i(\mathbf{u}_i, \boldsymbol{\theta}_i)\right)^2 d\mathbf{u}_i \tag{40}$$

subject to the equality constraint:

$$\Psi(\mathbf{x}, \boldsymbol{\theta}_i + \delta\boldsymbol{\theta}_i) - \Psi(\mathbf{x}, \boldsymbol{\theta}_i) = -\kappa_i \Delta y \tag{41}$$

where $1 \gg \kappa_i > 0$ is a user selected parameter which specifies the amount of error reduction in response to parameter adaptation.

Note that with a small positive number κ_i, the equation constraint (41) guarantees the prediction error to be reduced after the parameter is updated from $\boldsymbol{\theta}_i$ to $\boldsymbol{\theta}_i + \delta\boldsymbol{\theta}_i$ since

$$\begin{aligned} \Delta y_{\mathbf{new}}^{(i)} &\triangleq \Psi(\mathbf{x}, \boldsymbol{\theta}_i + \delta\boldsymbol{\theta}_i) - y \\ &= \Psi(\mathbf{x}, \boldsymbol{\theta}_i + \delta\boldsymbol{\theta}_i) - \Psi(\mathbf{x}, \boldsymbol{\theta}_i) + \Psi(\mathbf{x}, \boldsymbol{\theta}_i) - y \\ &= -\kappa_i \Delta y + \Delta y = (1 - \kappa_i)\Delta y \end{aligned} \tag{42}$$

which implies

$$\left\| \Delta y_{\mathbf{new}}^{(i)} \right\| = \left\| (1 - \kappa_i)\Delta y \right\| = \left\| (1 - \kappa_i) \right\| \left\| \Delta y \right\| < \left\| \Delta y \right\| \tag{43}$$

the above inequality holds since the inequality $1 \gg \kappa_i > 0$ (by assumption) implies $\left\| (1 - \kappa_i) \right\| < 1$.

The first-order approximate solution to the above-posed problem can be solved as stated in the following theorem.

Theorem 3 The first-order approximate solution of the parameter incremental $\delta\boldsymbol{\theta}_i$ that optimizes the performance index (40) subject to the equality constraint (41) is given by

$$\delta\boldsymbol{\theta}_i^{\mathbf{PIL}} \triangleq -\frac{\kappa_i}{\left(\dfrac{\partial \Psi(\mathbf{x}, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} \right)^{\mathbf{T}} \boldsymbol{\Theta}_i^{-1} \dfrac{\partial \Psi(\mathbf{x}, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i}} \boldsymbol{\Theta}_i^{-1} \frac{\partial \Psi(\mathbf{x}, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} \Delta y \tag{44}$$

Proof:

By using the first-order approximation of the $g_i(\mathbf{u}_i, \boldsymbol{\theta}_i + \delta\boldsymbol{\theta}_i)$ given by (33), the performance index (40) becomes

$$J_i\left(\boldsymbol{\theta}_i, \delta\boldsymbol{\theta}_i\right) \cong \int_{\mathbf{u}_i \in \mathfrak{D}_i} \left(\left(\frac{\partial g_i(\mathbf{u}_i, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} \right)^{\mathbf{T}} \delta\boldsymbol{\theta}_i \right)^{\mathbf{T}} \left(\left(\frac{\partial g_i(\mathbf{u}_i, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} \right)^{\mathbf{T}} \delta\boldsymbol{\theta}_i \right) d\mathbf{u}_i = \frac{1}{2} \delta\boldsymbol{\theta}_i^{\mathbf{T}} \boldsymbol{\Theta}_i \delta\boldsymbol{\theta}_i \tag{45}$$

And the equality constraint (41) becomes

$$\left(\frac{\partial \Psi(\mathbf{x}, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} \right)^{\mathbf{T}} \delta\boldsymbol{\theta}_i \cong -\kappa_i \Delta y \tag{46}$$

Let

$$\tilde{J}(\boldsymbol{\theta}_i, \delta\boldsymbol{\theta}_i) \triangleq \frac{1}{2} \delta\boldsymbol{\theta}_i^{\mathbf{T}} \boldsymbol{\Theta}_i \delta\boldsymbol{\theta}_i + \lambda \left(\left(\frac{\partial \Psi(\mathbf{x}, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} \right)^{\mathbf{T}} \delta\boldsymbol{\theta}_i + \kappa_i \Delta y \right) \tag{47}$$

where λ is the Lagrange multiplier. The optimal $\delta\boldsymbol{\theta}_i$ is then obtained by the following optimization condition:

$$\frac{\partial \tilde{J}\left(\boldsymbol{\theta}_i, \delta\boldsymbol{\theta}_i\right)}{\partial \delta\boldsymbol{\theta}_i} = \boldsymbol{\Theta}_i \delta\boldsymbol{\theta}_i + \lambda \frac{\partial \Psi(\mathbf{x}, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} = 0 \tag{48}$$

This leads to

$$\delta\boldsymbol{\theta}_i = -\lambda \boldsymbol{\Theta}_i^{-1} \frac{\partial \Psi(\mathbf{x}, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} \tag{49}$$

Substituting Eq. (49) into Eq. (46) yields

$$\lambda = \frac{\kappa_i \Delta y}{\left(\frac{\partial \Psi(\mathbf{x}, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} \right)^{\mathbf{T}} \boldsymbol{\Theta}_i^{-1} \frac{\partial \Psi(\mathbf{x}, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i}} \tag{50}$$

Substituting Eq. (50) back into Eq. (49) yields Eq. (44). This completes the proof of Theorem 3.

By comparing the Eq. (50) with the Eq. (29) it can be seen that they are essentially the same up to a selection of a free parameter, since both κ_i and λ_i are free parameters, and the Ω in Eq. (29) is a scalar for single output neural networks.

Chapter 4 Selection of parameters for the general PIL algorithm

One question remains to be solved in using the general PIL algorithm (at neuron level): the selection of the algorithm parameters λ_i, $i = 1, 2, \cdots, L$, and Ξ. This is clearly a non-trivial task. Apparently, it is unwise to have a fixed value for all λ_i. Unfortunately, there is no readily available theoretical guideline for selecting a set of parameters that could lead to satisfactory learning performance. From the practical point of view, however, a promising and convenient way of dealing with this issue is to compare the relevant properties of the PIL algorithm with that of the conventional stochastic gradient-descent algorithm since the latter works efficiently in many applications. In what follows, we will make a selection of the parameters of the PIL algorithm by *normalizing* neuron-by-neuron the error-reduction property of the PIL algorithm to that of the standard gradient descent learning algorithm.

The well-known *standard stochastic (or on-line) gradient-descent* learning algorithm is given by the following parameter update law:

$$\delta\boldsymbol{\theta}^{\mathbf{GRD}} \triangleq -\mu \frac{\partial E}{\partial \boldsymbol{\theta}} \tag{51}$$

where

$$E(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\Delta}\mathbf{y}^{\mathbf{T}}\boldsymbol{\Delta}\mathbf{y} \tag{52}$$

is the cost function for current training data $(\mathbf{x}, \mathbf{y})$; and $\mu > 0$ is the *learning rate*. Note that the spirit of the online gradient learning algorithm is that after the parameter update by Eq. (51), the cost function (52) will be (approximately) reduced, i.e.,

$$E(\boldsymbol{\theta} + \delta\boldsymbol{\theta}^{\mathbf{GRD}}) - E(\boldsymbol{\theta}) \cong \delta\boldsymbol{\theta}^{\mathbf{GRD}} \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = -\mu \left\| \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right\|^2 \leq 0 \tag{53}$$

To simplify the notation, let us denote the partial differentiation of $g_i(\mathbf{u}_i, \boldsymbol{\theta}_i)$, $\boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})$ with respect to the

parameter vector $\boldsymbol{\theta}_i$, the neuron g_i, respectively, as

$$\mathbf{p}_i(\mathbf{u}_i, \boldsymbol{\theta}_i) \triangleq \frac{\partial g_i(\mathbf{u}_i, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} \tag{54}$$

$$\mathbf{q}_i(\mathbf{x}, \boldsymbol{\theta}) \triangleq \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial g_i} \tag{55}$$

Then the following theorem holds.

Theorem 4 If the parameters for the PIL algorithm are chosen as

$$\Xi = \left(\varepsilon^{-1}\mathbf{I} - \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \boldsymbol{\Theta}_{\lambda}^{-1}(\boldsymbol{\theta}) \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{-1} \tag{56}$$

$$\frac{\varepsilon}{\lambda_i} = \mu \frac{\mathbf{p}_i^{\mathbf{T}} \mathbf{p}_i}{\mathbf{p}_i^{\mathbf{T}} \boldsymbol{\Theta}_i^{-1} \mathbf{p}_i}, \qquad i = 1, 2, \cdots, L \tag{57}$$

where ε is a sufficiently small positive number such that the Ξ given by (56) is guaranteed to be a positive-definite matrix. Then

$$E(\boldsymbol{\theta} + \delta\boldsymbol{\theta}^{\mathbf{GRD}}) \cong E(\boldsymbol{\theta} + \delta\boldsymbol{\theta}^{\mathbf{PIL}}) \tag{58}$$

Furthermore, the PIL algorithm is given by

$$\delta\boldsymbol{\theta}_i^{\mathbf{PIL}} = -\mu \frac{\mathbf{p}_i^{\mathbf{T}} \mathbf{p}_i}{\mathbf{p}_i^{\mathbf{T}} \boldsymbol{\Theta}_i^{-1} \mathbf{p}_i} \boldsymbol{\Theta}_i^{-1} \mathbf{p}_i \left(\mathbf{q}_i^{\mathbf{T}} \Delta\mathbf{y} \right), \quad i = 1, 2, \cdots, L \tag{59}$$

Proof:

Note that

$$\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_i} = \frac{\partial g_i(\mathbf{x}, \boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial g_i} \right)^{\mathbf{T}} = \mathbf{p}_i \mathbf{q}_i^{\mathbf{T}} \tag{60}$$

And

$$\frac{\partial E}{\partial \boldsymbol{\theta}} = \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \boldsymbol{\Delta}\mathbf{y} = \left[\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} \right)^{\mathbf{T}}, \quad \cdots, \quad \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_L} \right)^{\mathbf{T}} \right]^{\mathbf{T}} \boldsymbol{\Delta}\mathbf{y}$$
$$= \left[\mathbf{q}_1 \mathbf{p}_1^{\mathbf{T}}, \quad \cdots, \quad \mathbf{q}_L \mathbf{p}_L^{\mathbf{T}} \right]^{\mathbf{T}} \boldsymbol{\Delta}\mathbf{y} \tag{61}$$

Then

$$E(\boldsymbol{\theta} + \delta\boldsymbol{\theta}^{\mathbf{GRD}}) \cong E(\boldsymbol{\theta}) + \left(\frac{\partial E}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \delta\boldsymbol{\theta}^{\mathbf{GRD}}$$
$$= E(\boldsymbol{\theta}) - \mu \left(\frac{\partial E}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \frac{\partial E}{\partial \boldsymbol{\theta}} = E(\boldsymbol{\theta}) - \mu \boldsymbol{\Delta}\mathbf{y}^{\mathbf{T}} \left(\sum_{i=1}^{L} \left(\mathbf{p}_i^{\mathbf{T}} \mathbf{p}_i \right) \mathbf{q}_i \mathbf{q}_i^{\mathbf{T}} \right) \boldsymbol{\Delta}\mathbf{y} \tag{62}$$

On the other hand, note that with the Ξ given by Eq. (56), we have (from Eq. (31))

$$\Omega = \varepsilon \mathbf{I} \tag{63}$$

Then

$$E(\boldsymbol{\theta} + \delta\boldsymbol{\theta}^{\mathbf{PIL}}) \cong E(\boldsymbol{\theta}) + \left(\frac{\partial E}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \delta\boldsymbol{\theta}^{\mathbf{PIL}} = E(\boldsymbol{\theta}) - \boldsymbol{\Delta}\mathbf{y}^{\mathbf{T}} \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^{\mathbf{T}} \boldsymbol{\Theta}_{\lambda}^{-1}(\boldsymbol{\theta}) \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \Omega \boldsymbol{\Delta}\mathbf{y}$$

$$= E(\boldsymbol{\theta}) - \varepsilon \boldsymbol{\Delta}\mathbf{y}^{\mathbf{T}} \left[\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} \right)^{\mathbf{T}}, \quad \cdots, \quad \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_L} \right)^{\mathbf{T}} \right] \begin{bmatrix} \frac{\boldsymbol{\Theta}_1^{-1}}{\lambda_1} & & \\ & \ddots & \\ & & \frac{\boldsymbol{\Theta}_L^{-1}}{\lambda_L} \end{bmatrix} \begin{bmatrix} \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} \\ \vdots \\ \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_L} \end{bmatrix} \boldsymbol{\Delta}\mathbf{y}$$

$$= E(\boldsymbol{\theta}) - \varepsilon \boldsymbol{\Delta}\mathbf{y}^{\mathbf{T}} \left(\sum_{i=1}^{L} \frac{1}{\lambda_i} \left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_i} \right)^{\mathbf{T}} \boldsymbol{\Theta}_i^{-1} \frac{\partial \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_i} \right) \boldsymbol{\Delta}\mathbf{y}$$
$$= E(\boldsymbol{\theta}) - \boldsymbol{\Delta}\mathbf{y}^{\mathbf{T}} \left(\sum_{i=1}^{L} \frac{\varepsilon}{\lambda_i} \mathbf{p}_i^{\mathbf{T}} \boldsymbol{\Theta}_i^{-1} \mathbf{p}_i \mathbf{q}_i \mathbf{q}_i^{\mathbf{T}} \right) \boldsymbol{\Delta}\mathbf{y} = E(\boldsymbol{\theta}) - \mu \boldsymbol{\Delta}\mathbf{y}^{\mathbf{T}} \left(\sum_{i=1}^{L} \mathbf{p}_i^{\mathbf{T}} \mathbf{p}_i \mathbf{q}_i \mathbf{q}_i^{\mathbf{T}} \right) \boldsymbol{\Delta}\mathbf{y} \tag{64}$$
$$\cong E(\boldsymbol{\theta} + \delta\boldsymbol{\theta}^{\mathbf{GRD}})$$

This completes the proof of Eq. (58). The Eq. (59) is a straightforward result of substituting the equations (60), (63) and (57) into Eq. (29). This completes the proof.

Remark 1: Note that the Eq. (58) is equivalent to

$$E(\boldsymbol{\theta}+\delta\boldsymbol{\theta}^{\mathbf{PIL}})-E(\boldsymbol{\theta})\cong E(\boldsymbol{\theta}+\delta\boldsymbol{\theta}^{\mathbf{GRD}})-E(\boldsymbol{\theta}) \tag{65}$$

which implies that the quantity of the error-reduction in terms of the cost function (52) is approximately the same for the two algorithms for a given training pattern, despite that the learning algorithms differ. It is for this reason that the $\delta\boldsymbol{\theta}^{\mathbf{PIL}}$ is said to be *normalized* to the conventional gradient descent learning algorithm. Also we note that the normalization given by Eq. (57) is achieved neuron-by-neuron. The benefit from this fact is that, most likely, no re-tuning of the learning rate is needed when replacing the conventional on-line gradient descent learning algorithm with the PIL algorithm. It will be further confirmed in a numerical study on the PIL algorithm for MLP in Chapter 7 (Case 1), that a learning rate which works well for conventional on-line backpropagation learning algorithms also works well for the PIL algorithm of (59) (and vice versa).

Remark 2. The conventional stochastic gradient-descent learning algorithm, on the other hand, can be derived from the general PIL algorithm *at the parameter level*. In fact, the following index

$$\tilde{J}^{\,pres}=\frac{1}{2\mu}\boldsymbol{\theta}^{\mathbf{T}}\boldsymbol{\theta} \tag{66}$$

can be used as a measure of the preservation performance in the General PIL Problem. In this case, the measure of the preservation performance is against the "deformation" of the parameters instead of against the deformation of the neurons. This is the essential difference between the two parameter update algorithms. If the parameter Ξ is chosen as

$$\Xi=\left(\mathbf{I}-\mu\left(\frac{\partial\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right)^{\mathbf{T}}\frac{\partial\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right)^{-1} \tag{67}$$

then the solution to the general PIL problem at parameter level turns out to be

$$\delta\boldsymbol{\theta}_{*}^{\mathbf{PIL}}=-\mu\frac{\partial\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\Delta\mathbf{y} \tag{68}$$

which is precisely the same as the conventional gradient descent learning algorithm. In this sense, the conventional gradient descent learning algorithm can also be interpreted as a special case of the parameter incremental learning algorithm. The following corollary further clarifies the essential difference between the PIL algorithm and the conventional gradient descent learning algorithm.

Corollary 5 The following two inequalities hold

$$\left\|\delta\boldsymbol{\theta}_i^{\mathbf{GRD}}\right\| \le \left\|\delta\boldsymbol{\theta}_i^{\mathbf{PIL}}\right\| \qquad i = 1, 2, \cdots, L \tag{69}$$

$$\left\|\delta\boldsymbol{\theta}_i^{\mathbf{PIL}}\right\|_{\boldsymbol{\Theta}_i} \le \left\|\delta\boldsymbol{\theta}_i^{\mathbf{GRD}}\right\|_{\boldsymbol{\Theta}_i} \qquad i = 1, 2, \cdots, L \tag{70}$$

where

$$\delta\boldsymbol{\theta}_i^{\mathbf{GRD}} = -\mu\mathbf{p}_i\mathbf{q}_i^{\mathbf{T}}\boldsymbol{\Delta}\mathbf{y}, \qquad i = 1, 2, \cdots, L \tag{71}$$

is the neuron-wise form of the Eq. (51), $\delta\boldsymbol{\theta}_i^{\mathbf{PIL}}$ is given by the Eq. (59).

Proof:

Note that for any square positive-definite matrix $\mathbf{A}$, the following inequalities, which are direct results of using the Singular Value Decomposition for $\mathbf{A}$ followed by applying the Cauchy-Schwarz inequality, are well-known:

$$\frac{\mathbf{x}^{\mathbf{T}}\mathbf{A}^2\mathbf{x}}{\mathbf{x}^{\mathbf{T}}\mathbf{x}} \ge \left(\frac{\mathbf{x}^{\mathbf{T}}\mathbf{A}\mathbf{x}}{\mathbf{x}^{\mathbf{T}}\mathbf{x}}\right)^2 \tag{72}$$

and

$$\frac{\mathbf{x}^{\mathbf{T}}\mathbf{A}\mathbf{x}}{\mathbf{x}^{\mathbf{T}}\mathbf{x}}\frac{\mathbf{x}^{\mathbf{T}}\mathbf{A}^{-1}\mathbf{x}}{\mathbf{x}^{\mathbf{T}}\mathbf{x}} \ge 1 \tag{73}$$

where $\mathbf{x}$ is any non-zero vector with compatible dimension. Then

$$\frac{\left\|\delta\boldsymbol{\theta}_i^{\mathbf{PIL}}\right\|^2}{\left\|\delta\boldsymbol{\theta}_i^{\mathbf{GRD}}\right\|^2}=\frac{\left(\delta\boldsymbol{\theta}_i^{\mathbf{PIL}}\right)^{\mathbf{T}}\delta\boldsymbol{\theta}_i^{\mathbf{PIL}}}{\left(\delta\boldsymbol{\theta}_i^{\mathbf{GRD}}\right)^{\mathbf{T}}\delta\boldsymbol{\theta}_i^{\mathbf{GRD}}}=\frac{\mu^2\left(\dfrac{\mathbf{p}_i^{\mathbf{T}}\mathbf{p}_i}{\mathbf{p}_i^{\mathbf{T}}\boldsymbol{\Theta}_i^{-1}\mathbf{p}_i}\right)^2\mathbf{p}_i^{\mathbf{T}}\boldsymbol{\Theta}_i^{-2}\mathbf{p}_i\left(\mathbf{q}_i^{\mathbf{T}}\boldsymbol{\Delta}\mathbf{y}\right)^2}{\mu^2\mathbf{p}_i^{\mathbf{T}}\mathbf{p}_i\left(\mathbf{q}_i^{\mathbf{T}}\boldsymbol{\Delta}\mathbf{y}\right)^2}$$
$$=\frac{\mathbf{p}_i^{\mathbf{T}}\left(\boldsymbol{\Theta}_i^{-1}\right)^2\mathbf{p}}{\mathbf{p}_i^{\mathbf{T}}\mathbf{p}_i}\Bigg/\left(\frac{\mathbf{p}_i^{\mathbf{T}}\boldsymbol{\Theta}_i^{-1}\mathbf{p}_i}{\mathbf{p}_i^{\mathbf{T}}\mathbf{p}_i}\right)^2\geq 1 \tag{74}$$

and

$$\frac{\left\|\delta\boldsymbol{\theta}_i^{\mathbf{PIL}}\right\|_{\boldsymbol{\Theta}_i}^2}{\left\|\delta\boldsymbol{\theta}_i^{\mathbf{GRD}}\right\|_{\boldsymbol{\Theta}_i}^2}=\frac{\left(\delta\boldsymbol{\theta}_i^{\mathbf{PIL}}\right)^{\mathbf{T}}\boldsymbol{\Theta}_i\delta\boldsymbol{\theta}_i^{\mathbf{PIL}}}{\left(\delta\boldsymbol{\theta}_i^{\mathbf{GRD}}\right)^{\mathbf{T}}\boldsymbol{\Theta}_i\delta\boldsymbol{\theta}_i^{\mathbf{GRD}}}=\frac{\mu^2\left(\dfrac{\mathbf{p}_i^{\mathbf{T}}\mathbf{p}_i}{\mathbf{p}_i^{\mathbf{T}}\boldsymbol{\Theta}_i^{-1}\mathbf{p}_i}\right)^2\mathbf{p}_i^{\mathbf{T}}\boldsymbol{\Theta}_i^{-1}\mathbf{p}_i\left(\mathbf{q}_i^{\mathbf{T}}\boldsymbol{\Delta}\mathbf{y}\right)^2}{\mu^2\mathbf{p}_i^{\mathbf{T}}\boldsymbol{\Theta}_i\mathbf{p}_i\left(\mathbf{q}_i^{\mathbf{T}}\boldsymbol{\Delta}\mathbf{y}\right)^2}$$
$$=\frac{\mathbf{p}_i^{\mathbf{T}}\mathbf{p}_i}{\mathbf{p}_i^{\mathbf{T}}\boldsymbol{\Theta}_i^{-1}\mathbf{p}}\frac{\mathbf{p}_i^{\mathbf{T}}\mathbf{p}_i}{\mathbf{p}_i^{\mathbf{T}}\boldsymbol{\Theta}_i\mathbf{p}_i}\leq 1 \tag{75}$$

This completes the proof.

The first inequality in Corollary 5 means that the conventional gradient descent learning algorithm requires less parameter increment for each of the neurons than that of the PIL algorithm, given that both algorithms achieve approximately the same amount of error reduction in terms of the cost function E given by Eq. (52). On the other hand, the second inequality means that the PIL algorithm achieves a better preservation performance than that of the conventional gradient descent learning algorithm in terms of neuron deformation.

It is now clear that, in some sense, there are three levels of the general PIL problems: the global level, which is the highest level but difficult to obtain an analytical solution; the parameter level, which is the lowest level and its solution is basically the conventional gradient descent learning algorithm; and the neuron level (building block level), which is a mid-level between the highest and the lowest level and is the main concern in this work.

Remark 3: It is interesting to note that the PIL algorithm belongs to a more general class of gradient-descent

learning algorithms of the following form:

$$\delta\boldsymbol{\theta}^{\mathbf{GRD}} \triangleq -\mu\Phi\frac{\partial E}{\partial\boldsymbol{\theta}} \tag{76}$$

where Φ is an appropriately chosen positive definite symmetric matrix, $\mu > 0$ is a scalar learning rate, sometimes associated with an annealing procedure to ensure the final convergence phase of the algorithm. Typical of these algorithms are those so-called second-order methods [4] such as the Gauss-Newton method (or its modified version, the Levenberg-Marquardt method) where Φ is an approximation of the inversion of the Hessian matrix of $E(\boldsymbol{\theta})$, and the Natural Gradient method,[18] where Φ is the inverse of the Fisher Information matrix. With the selection of Ξ given by Eq. (56), the PIL expressed by Eq. (39) can be written as:

$$\delta\boldsymbol{\theta} = -\varepsilon\boldsymbol{\Theta}_{\lambda}^{-1}(\boldsymbol{\theta})\frac{\partial\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\Delta\mathbf{y} = -\varepsilon\boldsymbol{\Theta}_{\lambda}^{-1}(\boldsymbol{\theta})\frac{\partial E}{\partial\boldsymbol{\theta}} \tag{77}$$

where $\boldsymbol{\Theta}_{\lambda}(\boldsymbol{\theta})$ is a positive definite matrix (as is its inverse $\boldsymbol{\Theta}_{\lambda}^{-1}(\boldsymbol{\theta})$) since all the $\boldsymbol{\Theta}_i$'s are positive definite matrices by construction. This takes the form of (76) and is thus a general gradient descent method. However, unlike other general gradient descent methods where the inversion operation over a large matrix (with its dimension equal to the number of all adjustable parameters in the network) is required, the matrix $\boldsymbol{\Theta}_{\lambda}(\boldsymbol{\theta})$ is neuron-wise block-diagonal and thus the computational burden associated with the inversion operation is greatly reduced. In fact, as will be seen in Chapter 5, there is no need at all to do matrix inversion for the PIL algorithm for Multi-Layer Perceptron.

Chapter 5 PIL algorithm for Multi-Layer Perceptron

It is well-known that the most popular feedforward neural networks are the Multi-Layer Perceptron. Figure 1 shows an example of MLP structure with one hidden layer. In this chapter, the general PIL algorithm is applied to the derivation of the learning algorithm for the Multi-Layer Perceptron. Since the general PIL algorithm is expressed in the neuron-wise form, the following discussions in this section will be based on a representative neuron, *i.e.*, a neuron that can be any of the neurons in the neural network.

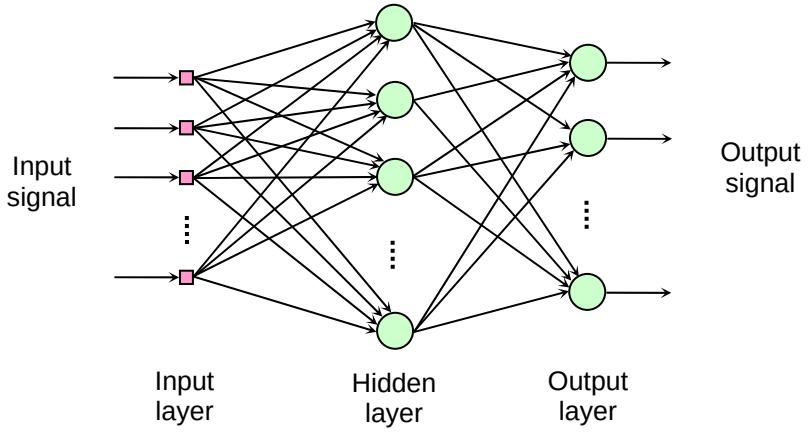

Figure 1 Multi-Layer Perceptron with one hidden layer

The nonlinear activation function of the neural network is chosen to be the most popular hyperbolic tangent function:

$$h(s) = \frac{1-e^{-s}}{1+e^{-s}} \tag{78}$$

In the MLP, the input to the activation function of a neuron is given by

$$s = \bar{\mathbf{u}}^{\mathbf{T}}\boldsymbol{\theta} \tag{79}$$

where $\bar{\mathbf{u}} = [\mathbf{1}, \mathbf{u}^{\mathbf{T}}]^{\mathbf{T}}$; $\mathbf{u} = [u_1 \quad u_2 \quad \cdots \quad u_n]^{\mathbf{T}} \in \Re^n$ is the input vector of the neuron; $\boldsymbol{\theta} = \begin{bmatrix}\theta_0 & \theta_1 & \cdots & \theta_n\end{bmatrix}^{\mathbf{T}}$ is the weight vector of this neuron. With the given activation function, a neuron in

the MLP is expressed as

$$y_s = g(\mathbf{u}, \boldsymbol{\theta}) = h(s) \tag{80}$$

where $y_s \in \Re$ is the output of the neuron. Then

$$\mathbf{p} = \frac{\partial g(\mathbf{u}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = h'(s)\bar{\mathbf{u}} \tag{81}$$

where

$$h'(s) = \frac{1}{2}(1 - h^2(s)) = \frac{2e^{-s}}{(1+e^{-s})^2} \tag{82}$$

then

$$\boldsymbol{\Theta}(\boldsymbol{\theta}) = \int_{\mathbf{u}\in\mathfrak{D}} \mathbf{p}(\mathbf{u}, \boldsymbol{\theta})\mathbf{p}^T(\mathbf{u}, \boldsymbol{\theta}) d\mathbf{u} = \int_{\mathbf{u}\in\mathfrak{D}} (h'(s))^2 \bar{\mathbf{u}}\bar{\mathbf{u}}^{\mathbf{T}} d\mathbf{u} \tag{83}$$

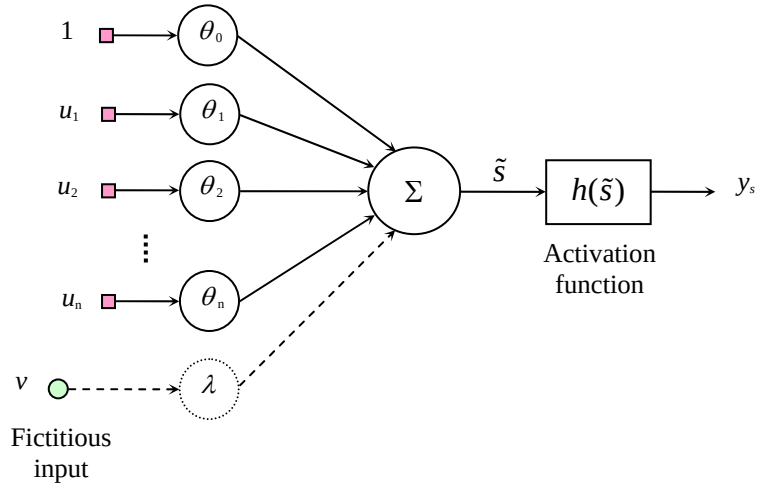

Figure 2 Model of a neuron augmented with fictitious input

The key issue in obtaining an analytical solution to the PIL problem is to obtain a "closed form" formula to the integral (83) with a reasonably defined region $\mathfrak{D}$. Unfortunately, this is generally not possible for this case. To overcome this difficulty, an extra fictitious input to the neuron is introduced, as is shown in Figure 2.

This amounts to re-writing the Eq. (80) as:

$$y_s = g(\tilde{\mathbf{u}}, \tilde{\boldsymbol{\theta}}) = h(\tilde{s}) \tag{84}$$

where $$\tilde{s} = \hat{\mathbf{u}}^{\mathbf{T}}\tilde{\boldsymbol{\theta}} \tag{85}$$

and $\hat{\mathbf{u}} = [1, \tilde{\mathbf{u}}^{\mathbf{T}}]^{\mathbf{T}}$; $\tilde{\mathbf{u}} = [\mathbf{u}^{\mathbf{T}}, v]^{\mathbf{T}}$ is the augmented input vector to the neuron; v denotes the fictitious input; $\tilde{\boldsymbol{\theta}} = [\boldsymbol{\theta}^{\mathbf{T}}, \lambda]^{\mathbf{T}}$ with λ being the weight corresponding to the fictitious input. Note that for the augmented neuron,

$$\tilde{\mathbf{p}} = \frac{\partial g(\tilde{\mathbf{u}}, \tilde{\boldsymbol{\theta}})}{\partial \tilde{\boldsymbol{\theta}}} = h'(\tilde{s}) \begin{bmatrix} 1 \\ \tilde{\mathbf{u}} \end{bmatrix} \tag{86}$$

The Eq. (83) is accordingly extended to:

$$\tilde{\boldsymbol{\Theta}}(\boldsymbol{\theta}, \lambda) = \int_{\tilde{\mathbf{u}} \in \tilde{\mathfrak{D}}} (h'(\tilde{s}))^2 \begin{bmatrix} 1 \\ \tilde{\mathbf{u}} \end{bmatrix} \begin{bmatrix} 1 & \tilde{\mathbf{u}}^{\mathbf{T}} \end{bmatrix} d\tilde{\mathbf{u}} \tag{87}$$

Now define the integral region as

$$\mathfrak{D} = \prod_{i=1}^{n} [-b_i/2,\ b_i/2], \quad b_i > 0 \ \text{ for } \ i = 1, 2, \cdots, n \tag{88}$$

and

$$\tilde{\mathfrak{D}} = \mathfrak{D} \times (-\infty, +\infty) \tag{89}$$

Then the PIL algorithm for MLP can be obtained as:

$$\delta\boldsymbol{\theta}^{\mathbf{PIL}} = -\mu h'(s) \frac{1 + \sum_{i=1}^{n} u_i^2}{1 + 12 \sum_{i=1}^{n} \frac{u_i^2}{b_i^2} + \frac{s^2}{\tau}} \begin{bmatrix} \frac{\theta_0 s}{\tau} + 1 \\ \frac{\theta_1 s}{\tau} + \frac{12}{b_1^2} u_1 \\ \vdots \\ \frac{\theta_n s}{\tau} + \frac{12}{b_n^2} u_n \end{bmatrix} \left(\mathbf{q}^{\mathbf{T}} \boldsymbol{\Delta}\mathbf{y} \right) \tag{90}$$

where $\tau = \dfrac{\pi^2 - 6}{3} \cong 1.29$. The derivation procedure is given in the Appendix A. Note that in some applications where a linear output neuron rather than a nonlinear activation function is preferred, *i.e.*, $g(s) = s$, it is straightforward to derive the PIL algorithm for those linear neurons, as given by

$$\delta\boldsymbol{\theta}^{\mathbf{PIL}} = -\mu \frac{1 + \sum_{i=1}^{n} u_i^2}{1 + \sum_{i=1}^{n} \dfrac{12}{b_1^2} u_i^2} \begin{bmatrix} 1 \\ \dfrac{12}{b_1^2} u_1 \\ \vdots \\ \dfrac{12}{b_n^2} u_n \end{bmatrix} \left(\mathbf{q}_\ell^{\mathbf{T}} \boldsymbol{\Delta}\mathbf{y} \right) \tag{91}$$

where $\mathbf{q}_\ell = \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}_{1\times m}^{\mathbf{T}}$. It is noted that for the linear neurons there would be no appreciable difference in using either the above PIL algorithm or the conventional gradient descent learning algorithm.

As a common selection, the integral region $\mathfrak{D}$ is set by $b_i = 2$ for all $i = 1, 2, \cdots, n$. The reasons for so doing are: Firstly, this selection is apparently reasonable for those neurons whose inputs are the outputs of other neurons from its preceding layer, since the output range of the neurons in the network is $(-1, +1)$; Secondly, for those neurons which receive the network inputs (*e. g.,* the neuron is in the first hidden layer), the range of input values can be readily normalized to $[-1, +1]$, as is a common practice in neural network learning.

Remark 4. The calculation of the $\mathbf{q}^{\mathbf{T}}\boldsymbol{\Delta}\mathbf{y}$ in Eq. (90) involves a well-known error-backpropagation procedure, thus the PIL algorithm can be considered as a variation of the standard BP algorithm. In fact, if the conventional gradient descent learning algorithm is re-written in the following neuron-wise form,

$$\delta\boldsymbol{\theta}^{\mathbf{BP}} = -\mu h'(s)\begin{bmatrix} 1 \\ u_1 \\ \vdots \\ u_n \end{bmatrix}\left(\mathbf{q}^{\mathbf{T}}\boldsymbol{\Delta}\mathbf{y}\right) \tag{92}$$

then the PIL algorithm seems to be only a “minor” modification of the conventional gradient descent learning algorithm. It is further noted from equation (90) that unlike other general gradient descent learning algorithms, the PIL algorithm requires no matrix inversion at all.

It is also worth noting that a similar learning algorithm for MLP with a logistic activation function can also be obtained by following a similar procedure in this section.

Chapter 6 Some theoretical relationships with the Natural Gradient Descent algorithm

As is introduced in the first chapter, the Natural Gradient Descent (NGD) learning algorithm is a principled stochastic learning algorithm for neural networks. The objective of this chapter is to reveal some intrinsic relationships between the GND and the PIL algorithms. First, it will be shown that the General PIL algorithm and the NGD algorithm, though they stem from different theoretical perspectives, are intrinsically related in that for a certain type of GND algorithm, the learning algorithm is essentially the same as PIL algorithm, up to the selection of a free parameter. Then, the analytic formulae for the simplest MLP, i.e., the MLP with a single non-linear neuron, are compared, revealing the similarity of the two algorithms, and the common basic feature that makes the two algorithms different from the standard BP algorithm.

6.1 The General PIL algorithm and the NGD of a certain type

Consider a stochastic feedforward neural network with a single output and additive Gaussian noise of the form:

$$y = \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta}) + \zeta \tag{93}$$

where $\zeta \sim N(0, \sigma^2)$ is the Gaussian additive noise with the variance σ^2. Then the likelihood function of y given $\mathbf{x}$ is expressed as

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{1}{2\sigma^2}\left[y - \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})\right]^2\right\} \tag{94}$$

The log likelihood function is therefore given by

$$\ell(y|\mathbf{x}, \boldsymbol{\theta}) = \log(p(y|\mathbf{x}, \boldsymbol{\theta})) = -\frac{1}{2\sigma^2}\left[y - \boldsymbol{\Psi}(\mathbf{x}, \boldsymbol{\theta})\right]^2 - \log\left(\sqrt{2\pi}\sigma\right) \tag{95}$$

It has been pointed out in [18] that in this case, the Riemann metric tensor $G(\boldsymbol{\theta})$ is given by the following Fisher information matrix,

$$G(\boldsymbol{\theta}) = E\left[\frac{\partial \ell(y|\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\left(\frac{\partial \ell(y|\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{\mathbf{T}}\right] \tag{96}$$

The partial differentiation of the log likelihood function with respect to the parameter vector $\boldsymbol{\theta}$ is expressed as

$$\frac{\partial \ell(y|\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{1}{\sigma^2}\left[y - \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})\right]\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{1}{\sigma^2}\zeta\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{97}$$

Note that

$$E(\zeta^2) = \sigma^2 \tag{98}$$

And assume that the input variable $\mathbf{x}$ has a distribution function $p(\mathbf{x})$ over a given region $\mathfrak{D}$. Then the Eq. (96) can be further written as

$$\begin{aligned} G(\boldsymbol{\theta}) &= E\left[\frac{1}{\sigma^4}\zeta^2\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{\mathbf{T}}\right] = \frac{E(\zeta^2)}{\sigma^4}E\left[\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{\mathbf{T}}\right] \\ &= \frac{1}{\sigma^2}E\left[\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{\mathbf{T}}\right] = \frac{1}{\sigma^2}\int_{\mathbf{x}\in\mathfrak{D}}\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{\mathbf{T}}p(\mathbf{x})d\mathbf{x} \end{aligned} \tag{99}$$

If it is further assumed that the input variable $\mathbf{x}$ has a uniform distribution over a given region $\mathfrak{D}$, i.e.,

$$p(\mathbf{x}) = \frac{1}{\mathrm{V}}, \quad \forall \mathbf{x} \in \mathfrak{D} \tag{100}$$

where V is a constant representing the hyper-volume of the region $\mathfrak{D}$. Then the Eq. (99) becomes

$$G(\boldsymbol{\theta}) = \frac{1}{\sigma^2 V}\int_{\mathbf{x}\in\mathfrak{D}}\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\left(\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^{\mathbf{T}}d\mathbf{x} = \frac{1}{\sigma^2 V}\boldsymbol{\Theta}(\boldsymbol{\theta}) \tag{101}$$

where the $\boldsymbol{\Theta}(\boldsymbol{\theta})$ is defined by Eq. (18) used by the General PIL algorithm. The NGD algorithm is thus given by

$$\begin{aligned}\delta\boldsymbol{\theta}^{NGD} &= -\eta G^{-1}(\boldsymbol{\theta})\frac{\partial \ell(y|\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \\ &= -\eta\left(\frac{1}{\sigma^2 V}\boldsymbol{\Theta}(\boldsymbol{\theta})\right)^{-1}\frac{1}{\sigma^2}\left[y-\boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})\right]\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = -(\eta V)\boldsymbol{\Theta}^{-1}(\boldsymbol{\theta})\frac{\partial \boldsymbol{\Psi}(\mathbf{x},\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\Delta y\end{aligned} \tag{102}$$

Comparing this NGD algorithm with the General PIL algorithm given by Eq. (16), it is seen that they are essentially the same up to a selection of a free parameter λ in Eq. (16) (also note that the Ω in Eq. (16) is a scalar for the single output case).

Therefore, it can be concluded that, although they stem from different theoretical perspectives, the NGD algorithm for the Gaussian stochastic feedforward neural network model with uniform distribution of input variable, and the General PIL algorithm, are essentially the same up to the selection of a free parameter.

6.2 The parameter update laws of the PIL and the NGD algorithms for a simple perceptron

One of the obstacles in applying the NGD algorithm to the feedforward neural networks is the lack of an analytical solution for the Riemann metric tensor matrix. To overcome the difficulty, [20] proposed an adaptive method of directly obtaining an approximation of the inverse of Fisher information matrix by applying the Kalman filter technique; and the computational cost is reduced to the order of $O(n^2)$ operations thereby, instead of the $O(n^3)$ if the Fisher information matrix has to be numerically inverted. The only analytical solution of the NGD algorithm for neural networks obtained so far, to the best knowledge of the author, is the one presented in [18] for a simple (non-linear) perceptron, or the simplest MLP with one layer: a single, non-linear neuron. It is interesting to make a comparison between the parameter update laws for the two algorithms, PIL and NGD, for this simple case, so that some insights can be gained.

Assume that the MLP has only one neuron with a hyper-tangent activation function $h(s)$ given by Eq. (78) in Chapter 5. The stochastic model of the perceptron considered in [18] is given by

$$y = h(s) + \xi \tag{103}$$

where

$$s = \mathbf{\theta}^{\mathbf{T}}\mathbf{x} \tag{104}$$

is the scalar input to the activation function; $\xi \sim N(0, \sigma^2)$ is the Gaussian additive noise with the variance σ^2. It is noted that there is no input bias term in this perceptron. It is further assumed that the input variable has a Gaussian distribution with identity variance matrix, i.e., $\mathbf{x} \sim N(0, \mathbf{I})$.

Under the above conditions, [18] proved that, for the training input/output pattern $(\mathbf{x}, y)$, the parameter update law of the NGD algorithm can be explicitly expressed as:

$$\delta\mathbf{\theta}^{\mathbf{NGD}} = -\mu\left(\kappa_1(\mathbf{\theta})\mathbf{x} + \kappa_2(\mathbf{\theta})s\mathbf{\theta}\right)h'(s)\Delta y \tag{105}$$

where $\kappa_1(\mathbf{\theta})$ and $\kappa_2(\mathbf{\theta})$ are known scalar functions of the parameter vector $\mathbf{\theta}$; $\Delta y = y - h(\mathbf{\theta}^{\mathbf{T}}\mathbf{x})$ is the prediction error for the current training pattern; μ is the learning rate.

On the other hand, from the Eq. (90) in Chapter 5, the parameter update law of the PIL algorithm for this case is can be written as

$$\delta\mathbf{\theta}^{\mathbf{PIL}} = -\mu\left(\tau_1(\mathbf{\theta}, \mathbf{x})\mathbf{x} + \tau_2(\mathbf{\theta}, \mathbf{x})s\mathbf{\theta}\right)h'(s)\Delta y \tag{106}$$

where $\tau_1(\mathbf{\theta}, \mathbf{x})$ and $\tau_2(\mathbf{\theta}, \mathbf{x})$ are known scalar functions of the parameter vector $\mathbf{\theta}$ and input vector $\mathbf{x}$.

By comparing the parameter update laws of the NGD algorithm (105) and the PIL algorithm (106), it can be observed that they share a common feature, which is remarkably different from the standard BP algorithm, in that the parameter vector incremental direction is a linear combination of the input vector $\mathbf{x}$ and the parameter vector $\mathbf{\theta}$, with the latter being proportional to the scalar input to the activation function of the perceptron (i.e., the s given by Eq. (104)). While for the standard BP algorithm, the parameter vector $\mathbf{\theta}$ increases along precisely the direction of the input vector $\mathbf{x}$, i.e.,

$$\delta\boldsymbol{\theta}^{\mathbf{BP}} = -\mu\mathbf{x}h'(s)\Delta y \tag{107}$$

This observation reveals the intrinsic feature of the PIL and NGD algorithms that distinguishes them from the standard BP algorithm.

Chapter 7 Numerical experiments – comparison with standard BP algorithm

Computer simulations on eight benchmark problems were carried out to study the performance of the PIL algorithm, mainly by comparing it with the standard on-line BP algorithm. To this end, a MATLAB® Simulink library block which implemented both the PIL and the standard BP algorithm was developed, as is shown in Figure 3. In the figure, the MLP Neural Network was programmed as an S-function using C programming language. The C source code is listed in Appendix B.

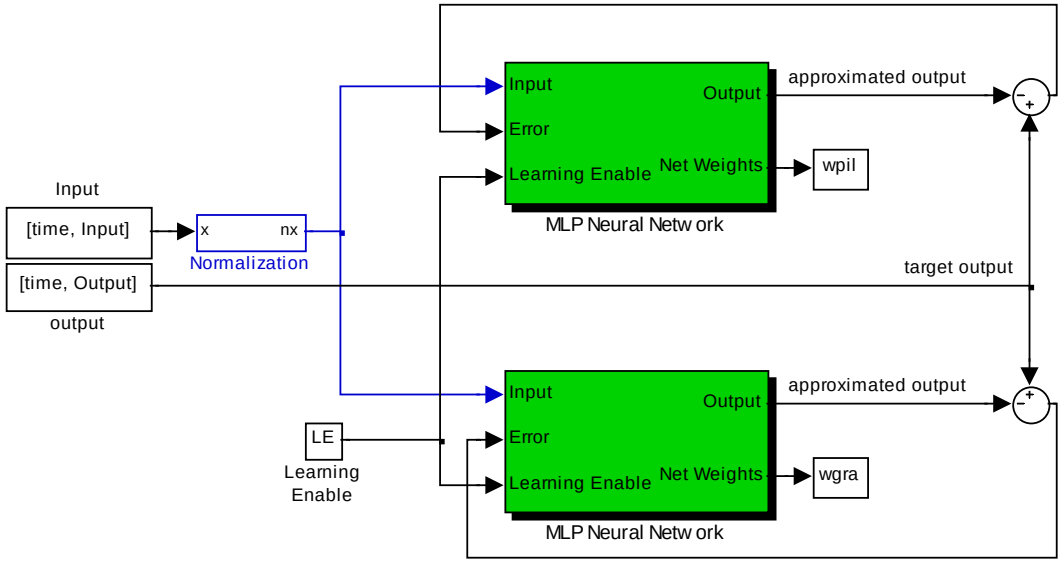

Figure 3 Simulink block diagram of the comparison study

In the computer simulation, some of the conventions in MLP training are used. These include:

1) The weights to each of the neurons are all initialized with uniformly distributed random values in [-1/m, 1/m] where the m is the fan-in (*i.e.*, number of inputs) to the neuron.
2) For each training epoch, the data is shuffled (randomly permuted) before it is used for training.
3) The input data range is normalized to be in the interval [-1, +1].

Also the following conditions were set in the numerical study.

1) All the MLPs used have one hidden layer with hyper-tangent sigmoid activation function.

2) For each of the learning tasks, two MLP neural networks with exactly the same structure (*i.e.*, the same number of hidden neurons) but using different learning algorithms (PIL vs. standard gradient descent), will be running in parallel with the same initial conditions (*i.e.*, initial weights). This means that both networks receive the same sequence of the input/output patterns during the training process.
3) The learning process will be visualized by the *learning curve, i.e.*, the plot of "MSE (Mean Square Error) vs. number-of-epochs", and the plot of "classification error vs. number-of-epochs" if it is a classification problem.
4) The output layer is chosen to be non-linear (with the same activation function as for the hidden neurons) for the classification task, and linear for the function approximation task.
5) For classification problems, the popular 40-20-40 criterion is used: an output is considered to be a logical zero if it is in the lower 40% of the output range, a one if it is in the upper 40%, and indeterminate (and therefore incorrect) if it is in the middle 20% of the range. The desired value logic one (zero) was chosen to be 90% (10%) of the output range to the maximum (minimum) output value. Since for an output neuron, the output range is (-1, 1), the desired logic one (zero) value is therefore 0.8 (-0.8).

7.1 Case 1: Peaks function approximation

The task is to approximate the following MATLAB® Peaks function using MLP:

$$z = 3(1-x)^2 e^{\left(-x^2-(y+1)^2\right)} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{\left(-x^2-y^2\right)} - \frac{1}{3}e^{\left(-(x+1)^2-y^2\right)} \tag{108}$$

The function mainly consists of a couple of peaks as shown in Figure 4. The training data consists of 3721 input-output data points which were obtained by evaluating the function (108) over the 61×61 uniformly distributed grid on the square $[-3,+3]\times[-3,+3]$ in the $x-y$ plane. Nine simulations were conducted, with

10, 15 and 20 hidden neurons, with each network trained using three different learning rates, 0.001, 0.002 and 0.003. The learning curves are shown from Figure 6 through Figure 14. Each training started from different random initial weights. It is seen that the PIL algorithm gives better performance than the conventional gradient descent learning algorithm in all cases in terms of convergence speed. Particularly, the PIL algorithm is less likely to get stuck in a so-called *flat area* or *plateau*, which is common in conventional BP algorithm and is one of the major reasons for the BP being criticized as slow in convergence. It can also be concluded from the nine simulations, as was explained in Remark 1 in Chapter 4, that a learning rate that is well-tuned for the standard BP algorithm is also well-tuned for the PIL algorithm, and vice versa. Figure 5 shows the approximated Peaks function by MLP, trained with PIL algorithm, with 20 hidden neurons.

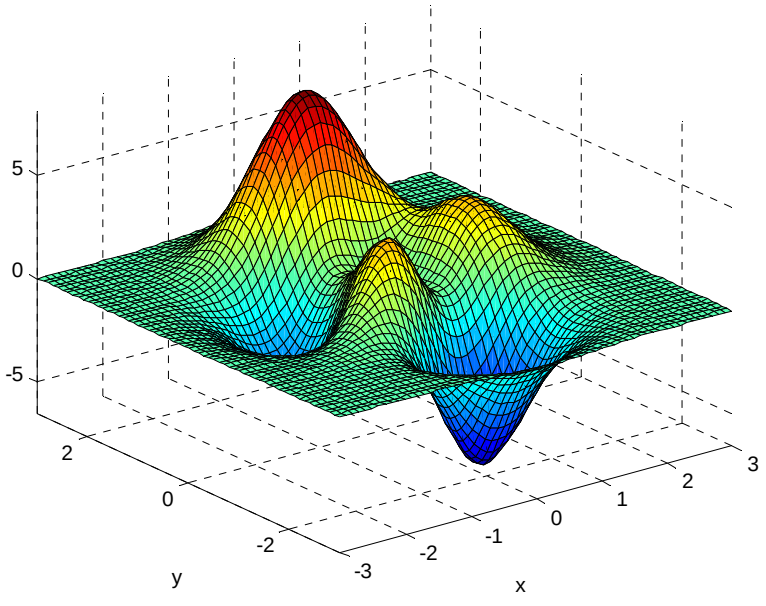

Figure 4 Peaks function

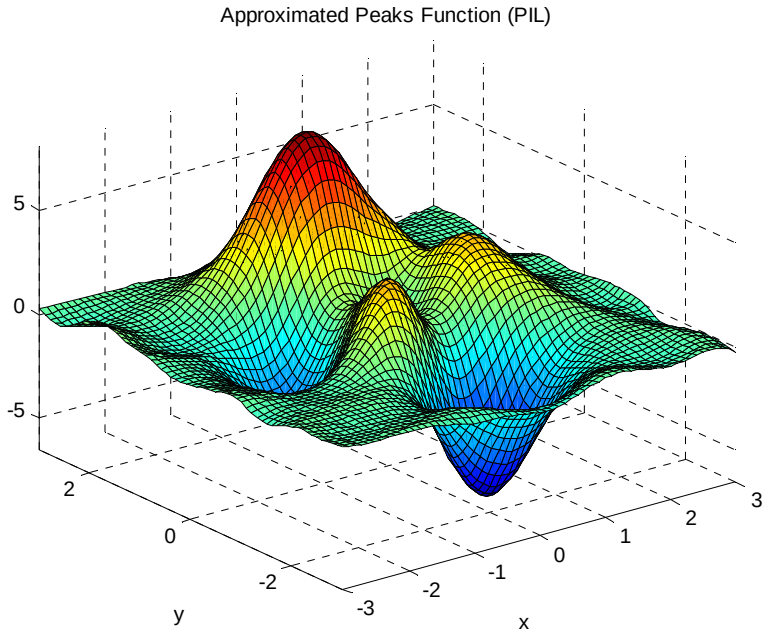

Figure 5 Approximated Peaks Function by MLP trained with PIL algorithm

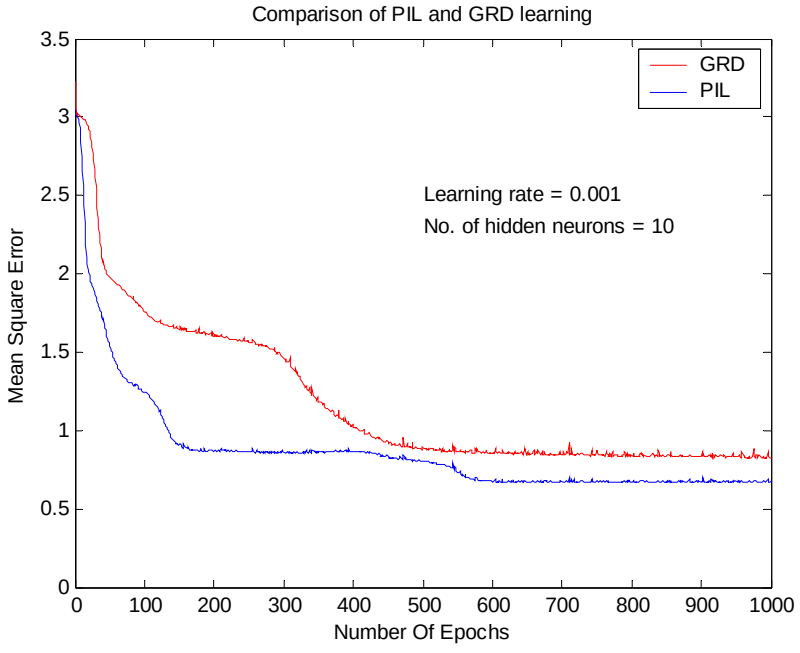

Figure 6 Learning curves for Peaks function

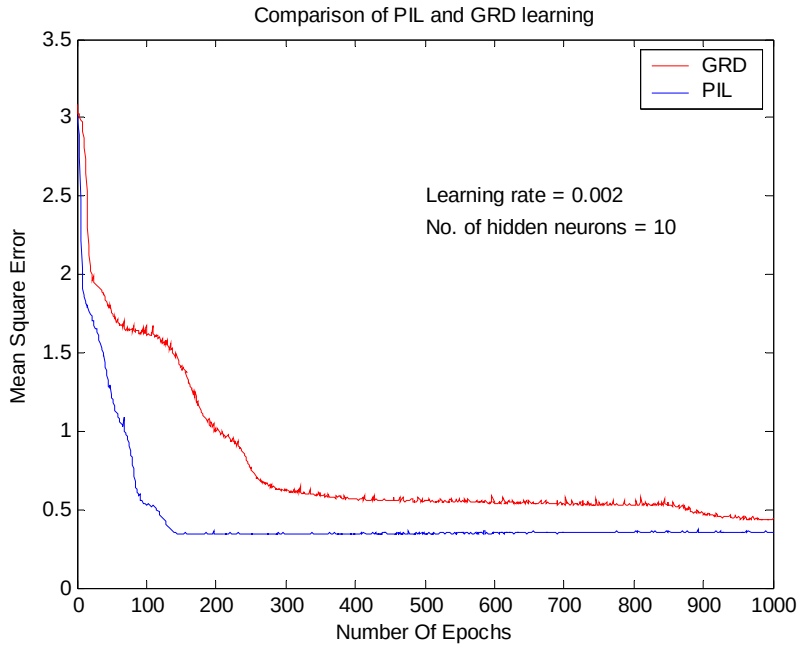

Figure 7 Learning curves for Peaks function

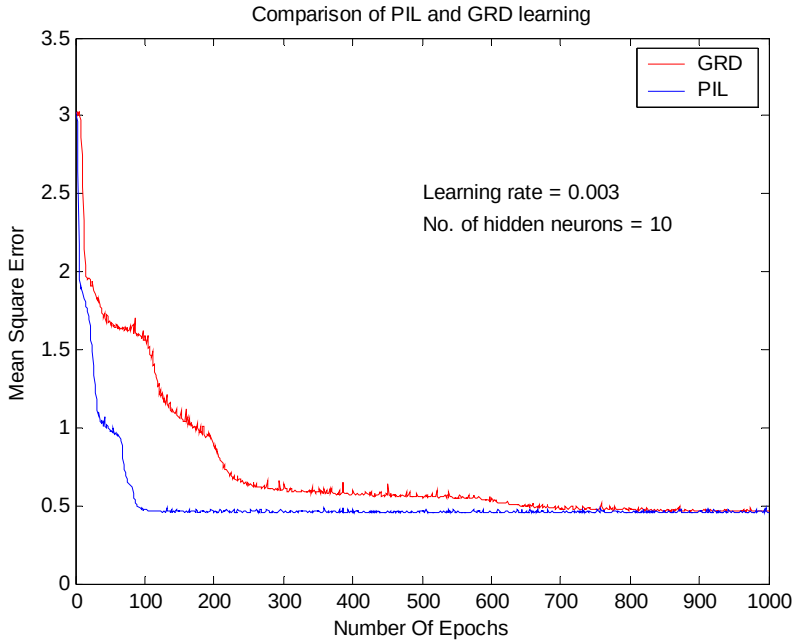

Figure 8 Learning curves for Peaks function

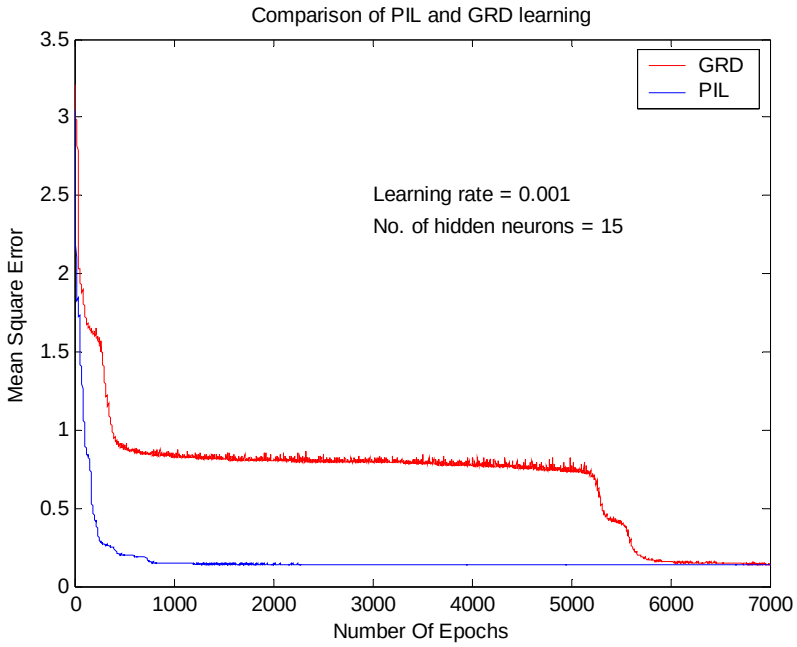

Figure 9 Learning curves for Peaks function

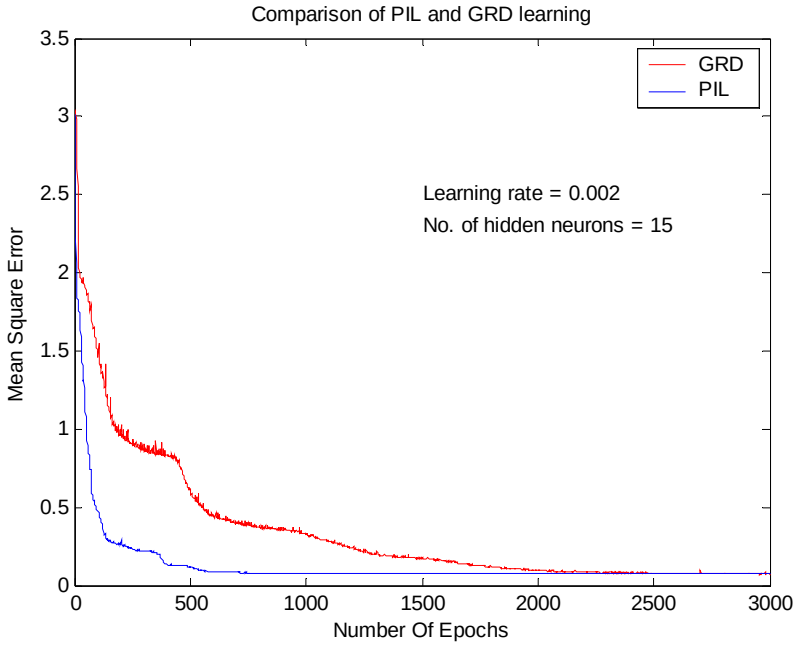

Figure 10 Learning curves for Peaks function

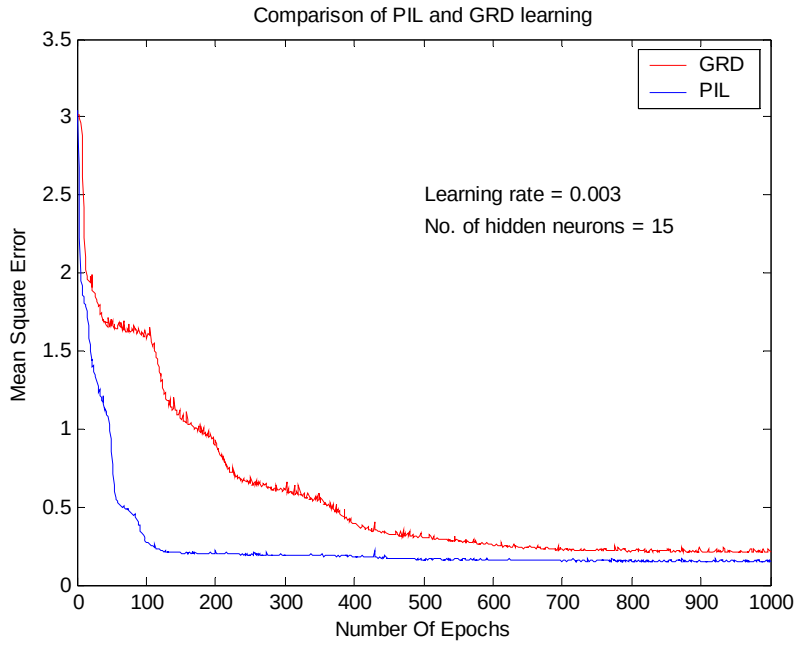

Figure 11 Learning curves for Peaks function

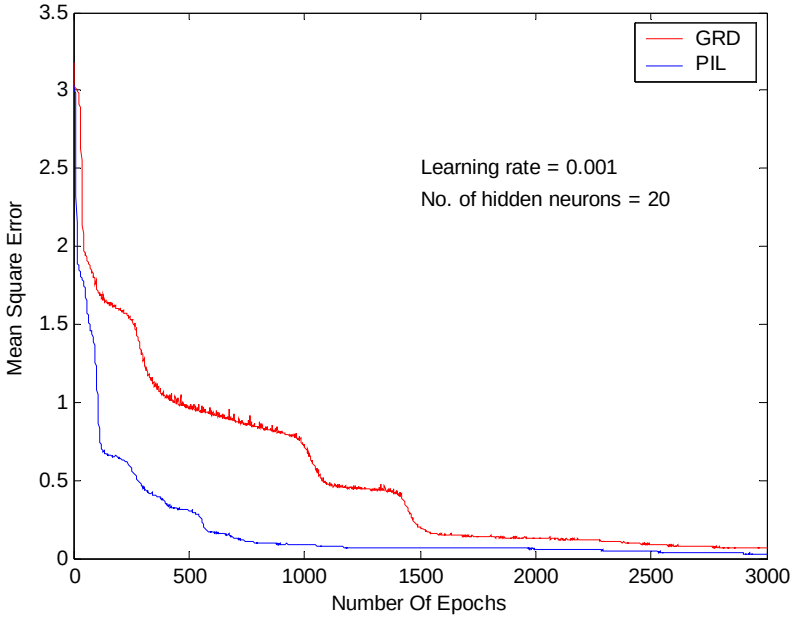

Figure 12 Learning curves for Peaks function

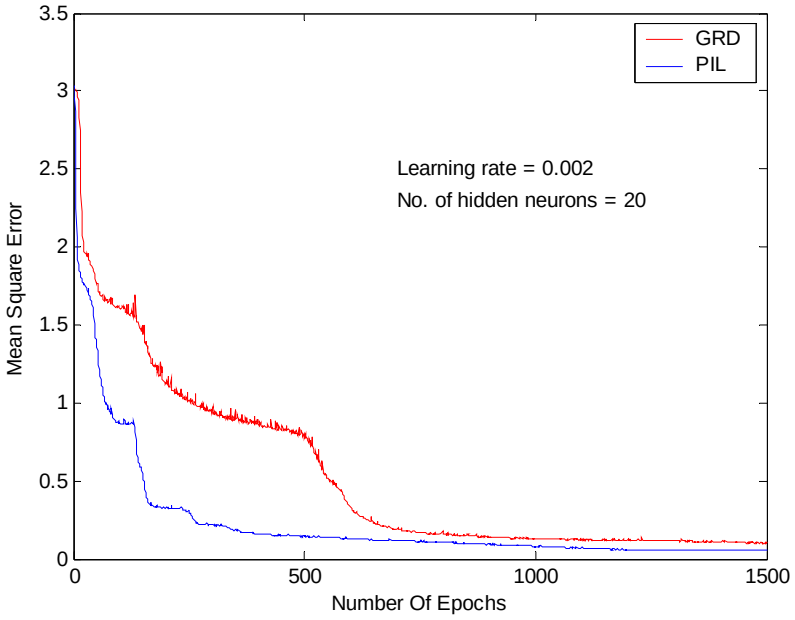

Figure 13 Learning curves for Peaks function

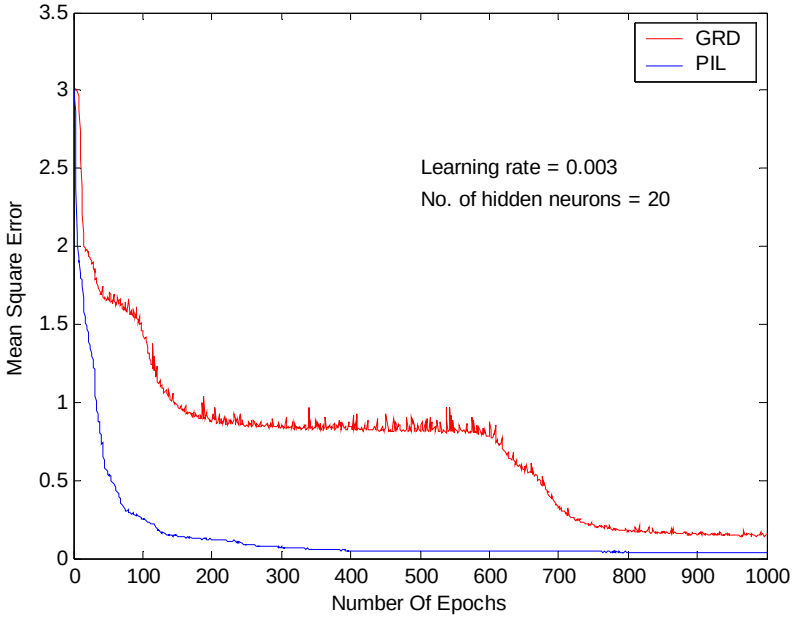

Figure 14 Learning curves for Peaks function

7.2 Case 2: Gabor function approximation

The task is to approximate the following 2-D Gabor function using MLP, shown in Figure 15:

$$z = \frac{2}{\pi} e^{\left(-2\left(x^2+y^2\right)\right)} \sin\left(2\pi\left(x+y\right)\right) \tag{109}$$

The training data consists of 3721 input-output data points which were obtained by evaluating the function (109) over the 61×61 uniformly distributed grid on the square $\left[-1.5, +1.5\right]\times\left[-1.5, +1.5\right]$ in the $x-y$ plane. Two simulations were conducted, where 2 different numbers of hidden neurons, 10 and 15, were used. The learning rate was 0.001 for both trainings. The learning curves are shown in Figure 17 and Figure 18. In both trainings the PIL algorithm can learn (to a certain degree of precision) the function, though it initially gets stuck on a long plateau. The conventional gradient descent learning algorithm completely fails to learn the function for both trainings; even though the number of epochs reaches as high as 200,000. Figure 16 shows the approximated Gabor function by MLP, trained with PIL algorithm, with 15 hidden neurons.

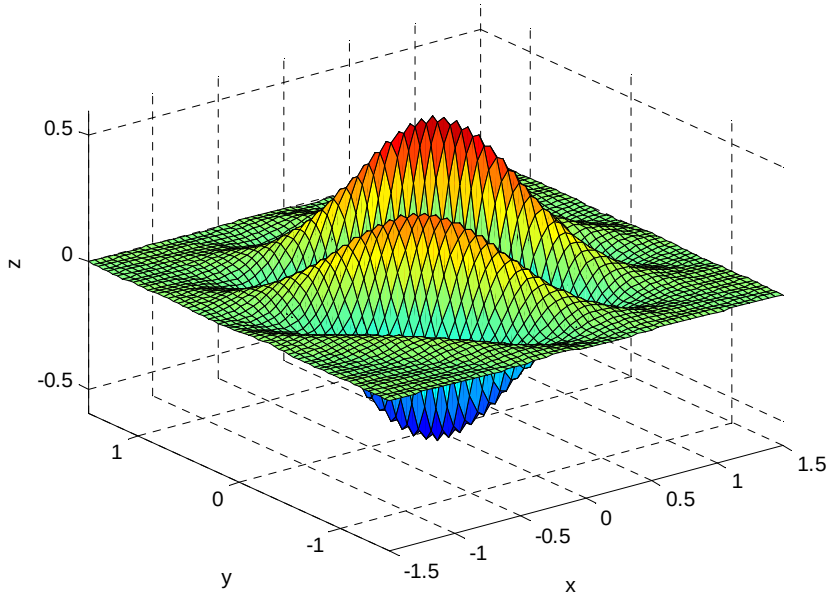

Figure 15 Gabor function

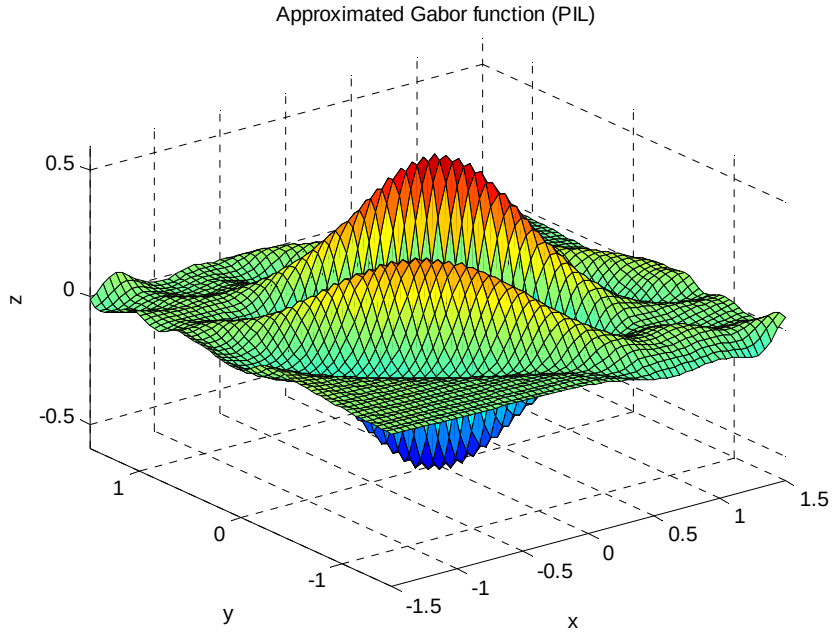

Figure 16 Approximated Gabor Function by MLP trained with PIL algorithm

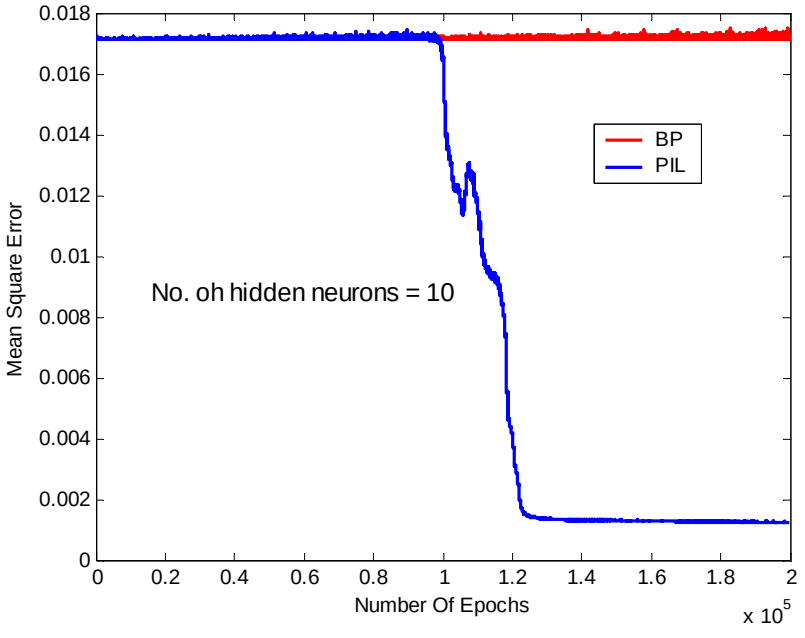

Figure 17 Learning curves for Gabor function

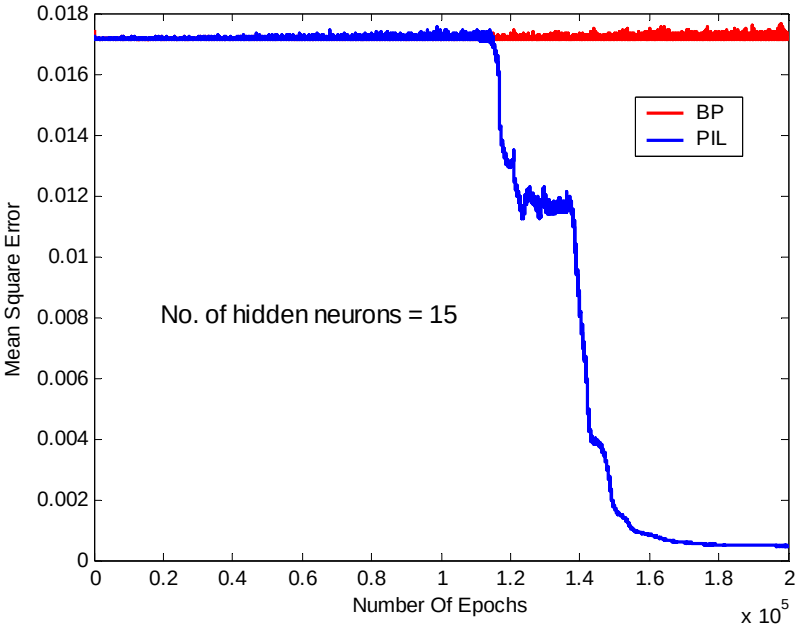

Figure 18 Learning curves for Gabor function

7.3 Case 3: Two spirals classification

The task is to learn to discriminate between two sets of training points which lie on two distinct spirals in the x-y plane, as is shown in Figure 19. These spirals coil three times around the origin and around one another. This appears to be a very difficult classification task for standard MLP since as was reported in [21] that standard back-propagation learning has been reported to fail completely when using a MLP with single hidden layer. A 3-hidden layer MLP, with 5 neurons in each and shortcut connections between all the layers, was found to be able to solve this hard problem. The two-spiral problem has since been popular in the neural network community and has been extensively used as a benchmark for evaluating the classification performance of various neural networks with different learning algorithms and/or structures.

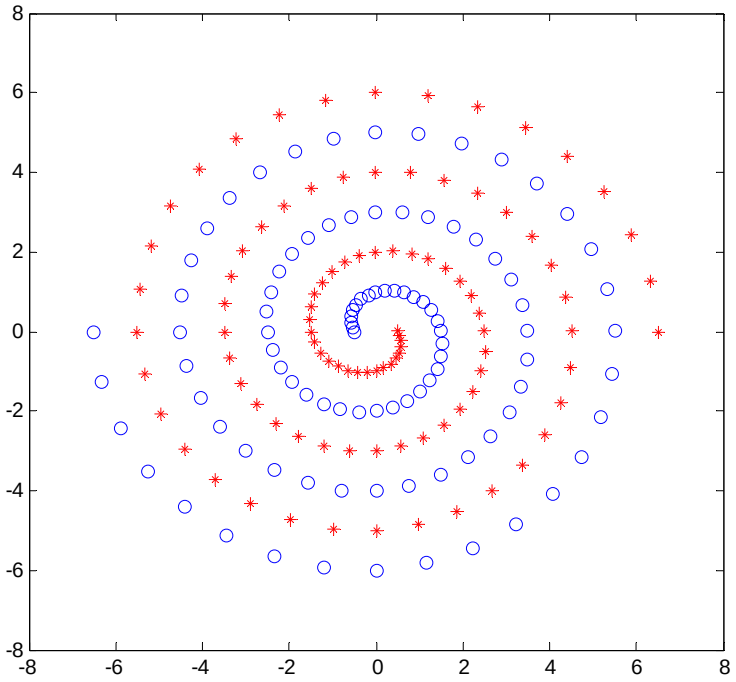

Figure 19 Two spirals

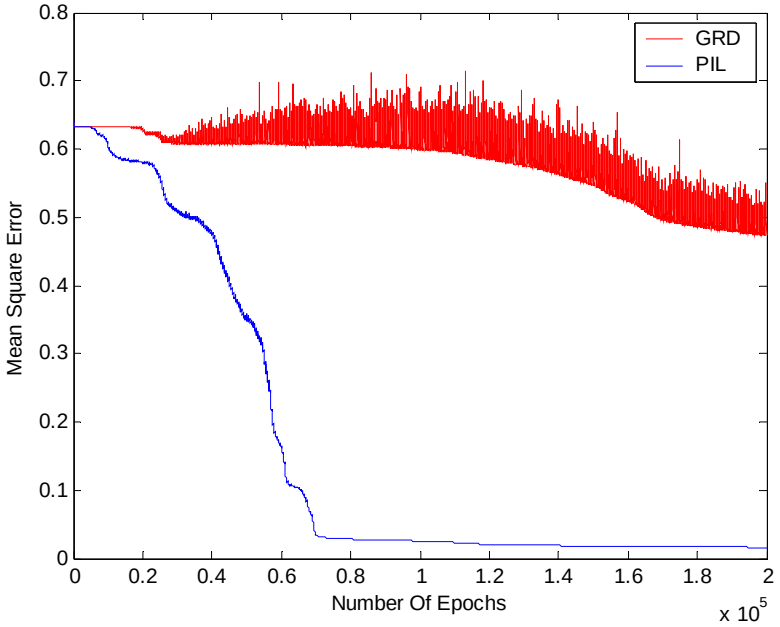

Figure 20 Learning curves for Two Spirals - MSE

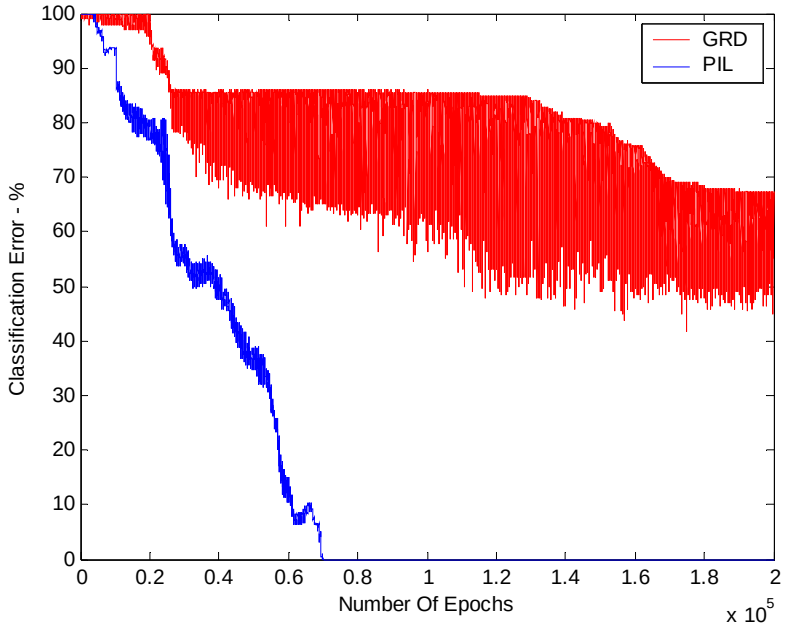

Figure 21 Learning curves for Two Spirals – Classification Error

The training data set consists of 194 input-output pairs, which is available at [22]. Forty five hidden neurons were used for this task. The learning rate was set to be 0.004. Figure 20 and Figure 21 show the learning curves of MSE *vs*. number of epochs and classification error *vs*. number of epochs, respectively. It is seen that the PIL algorithm learned the task with zero classification error after around 70,000 epochs, while the conventional gradient descent learning algorithm (standard BP), as was expected, completely failed even after 200,000 epochs.

7.4 Case 4: Ridges-Bump function approximation

The task is to approximate the function:

$$z = \max\left\{e^{-10x^2},\quad e^{-50y^2},\quad 1.25e^{-5(x^2+y^2)}\right\} + \frac{e^{-(x^2+y^2)/2}}{\sqrt{2\pi}} \tag{110}$$

This function is taken from [23] where it was used as a benchmark for evaluating the performance of a learning algorithm for neural networks. The function consists of a narrow and a wide ridge which are perpendicular to each other, and a Gaussian bump at the origin, as is shown in Figure 22. The training data consists of 1681 input-output data points which were obtained by evaluating the function (110) over the 41×41 uniformly distributed grid over the unit square in the x-y plane.

In the simulations, the number of hidden neurons was set to 15, and the learning rate was chosen as 0.001. The number of training epochs was set to 2×10^5. Two networks, denoted by A and B, with exactly the same network structure and initial weights, are trained in parallel but using different algorithms, PIL (for network A) and gradient descent (GRD, or standard BP), respectively. From Figure 24 it is seen that network B (with standard BP algorithm) got stuck on a long plateau, while the network A (with PIL learning algorithm) converges reasonably fast. To further demonstrate that the PIL has a better chance to avoid the plateau, the network B, which got stuck at a plateau when using standard BP, was re-trained starting from somewhere in the middle of the plateau, *i.e.*, the network's weights were re-initialized to the values obtained at the point of 10^5 training epochs, but the learning algorithm is switched (from conventional BP) to the PIL algorithm. An interesting phenomenon is that instead of following its previous learning curve, the network B starts to converge (toward the learning curves of network A where the PIL algorithm was used) faster than it had before, demonstrating the advantage of the new learning algorithm. Figure 23 shows the approximated Ridges-Bump function by MLP trained with the PIL algorithm.

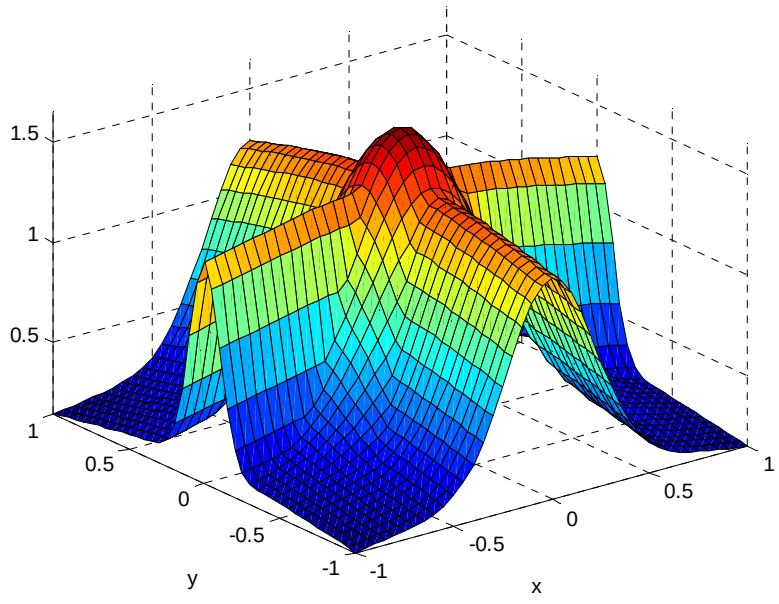

Figure 22 Ridges-Bump function

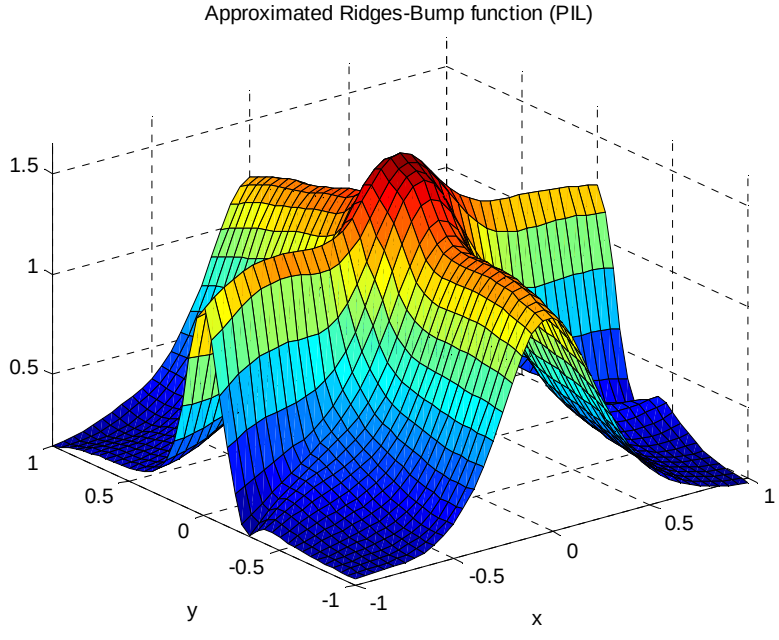

Figure 23 Approximated Ridges-Bump function by MLP trained with PIL algorithm.

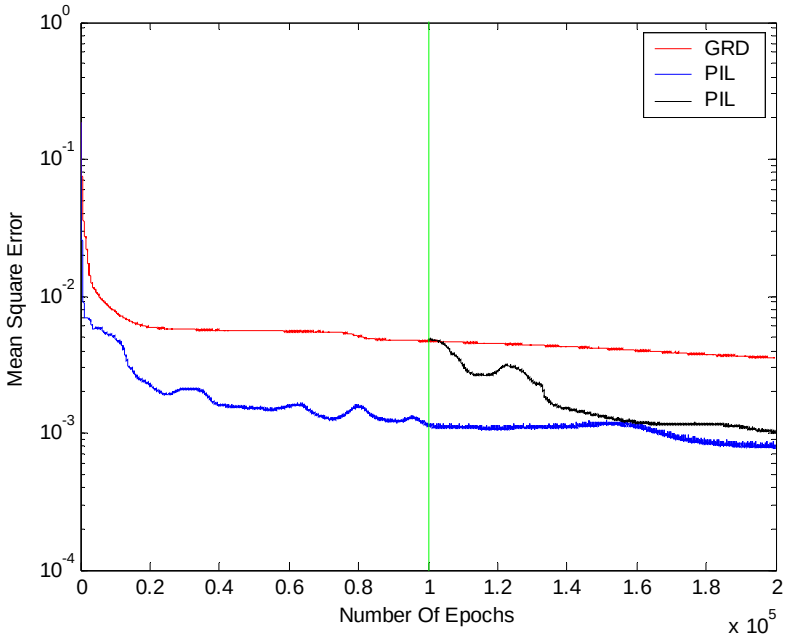

Figure 24 Learning curves for Ridges-Bump function

7.5 Case 5: Iris flower

The classification of irises is a famous benchmark problem in pattern recognition. Fisher used the data set in his classic paper on discriminant analysis [24] and the data set has since become a favorite example in pattern recognition (perhaps it is the best known database to be found in the pattern recognition literature). The data set, available from [25], contains 3 classes of 50 instances each, where each class refers to a type of iris plant. Each sample possesses four attributes. One class is linearly separable from the other two, but the latter are not linearly separable from each other, as is seen from Figure 25 where the 3 classes are shown on the four 3-D spaces by projecting the 4-dimensional data space onto four 3-D sub-spaces. The task is to determine to which of the three species a specimen belongs, given the four parameters.

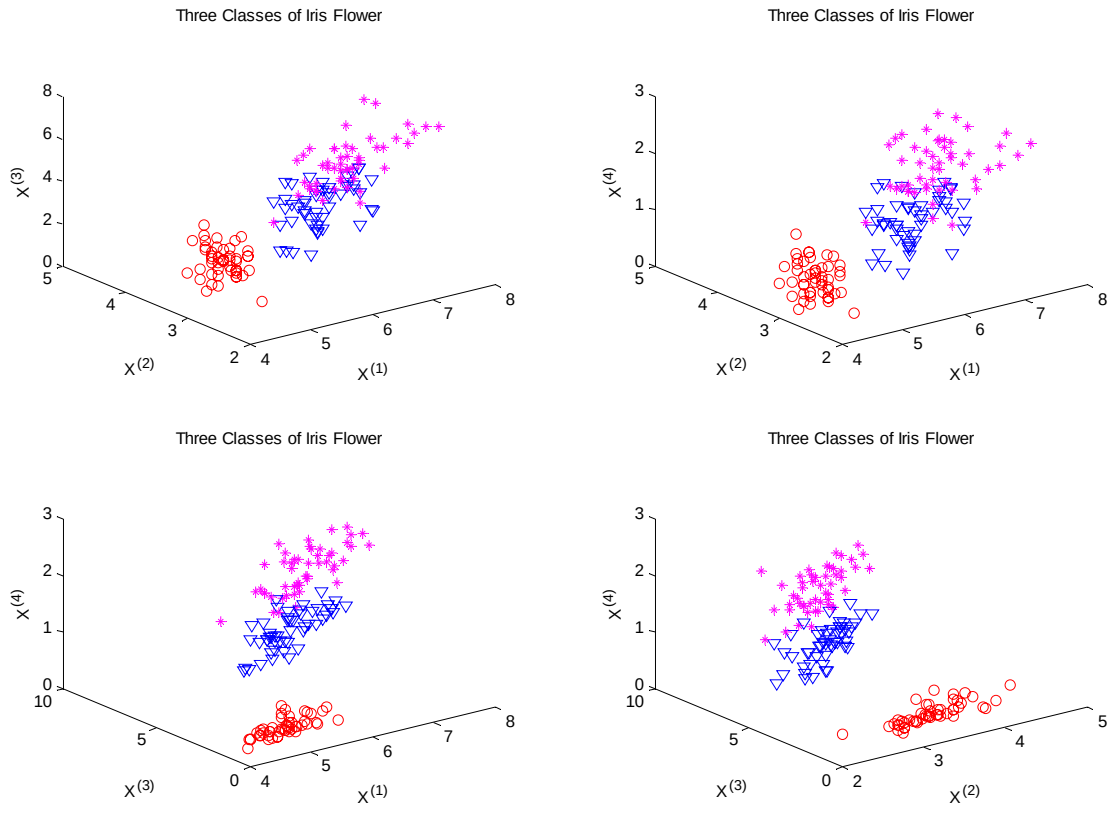

Figure 25 Three classes of iris flower projected in 3-D spaces

In the simulations, the number of neural network outputs is set to 3, with each corresponding to a class of iris. Two hidden neurons were used. The learning rates are all set to 0.05 for both networks (with standard BP and PIL algorithms), since it was found that this learning rate is well-tuned for both networks.

Ten runs were conducted, with the number of epochs being set to 40,000 for all runs since it was observed that this number of epochs is sufficient for both networks to reach a steady state classification error. It was found that in all the 10 runs both networks can reach a minimum classification error of only 1 misclassification out of the 150 samples. However, the number of training epochs needed to achieve this optimal classification performance differs significantly for the two networks, as is shown in the Table 1. It is seen that the average number of epochs needed for the standard BP algorithms (i.e., 34711) is about sixty

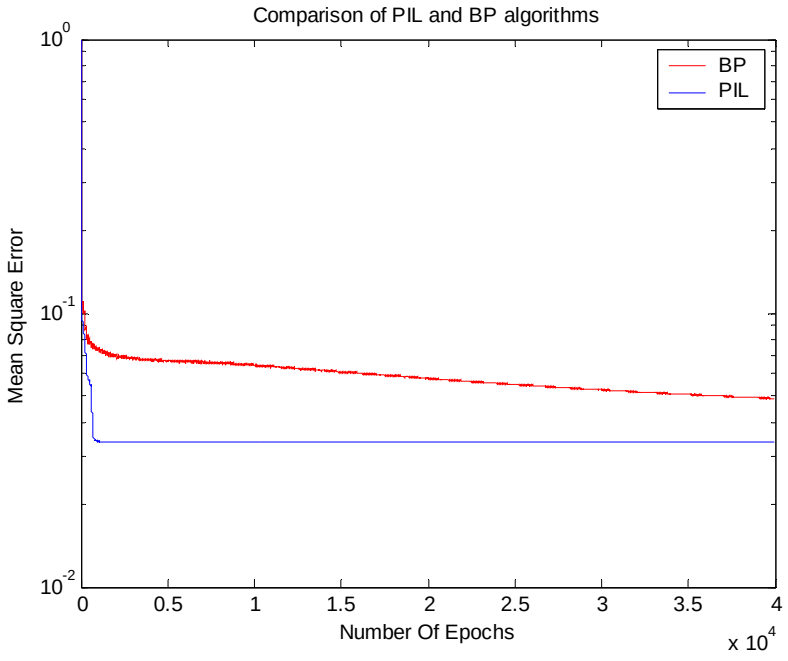

Figure 26 A typical learning curves for Iris flower classification - MSE

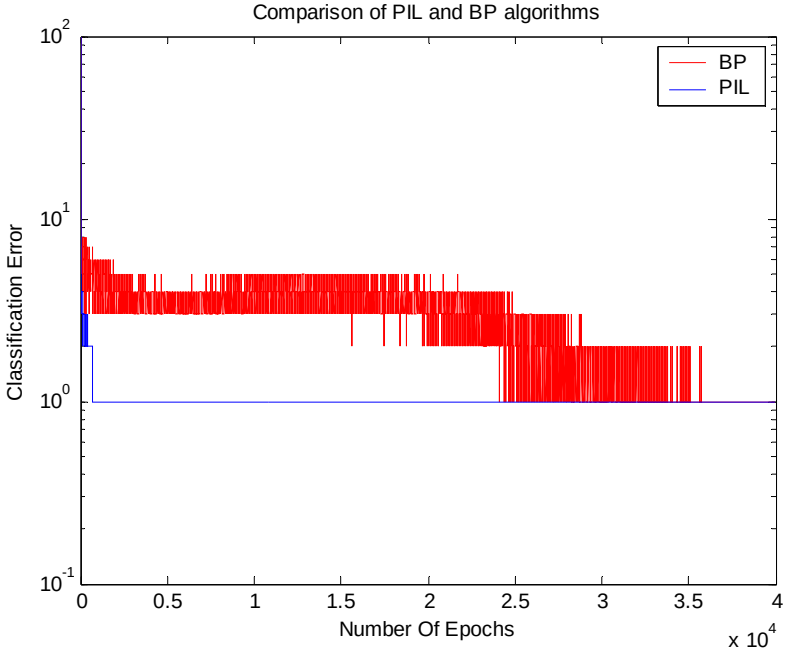

Figure 27 A typical learning curves for Iris flower classification – classification error

times larger than that needed by PIL algorithm (i.e., 579), demonstrating the remarkable advantage of the new algorithm. Figure 26 and Figure 27 show the typical learning curves of MSE *vs*. Number of Epochs and Classification Error *vs*. Number of Epochs, respectively. It is seen that the PIL algorithm is superior to the standard BP algorithm in terms of convergence speed.

Table 1 Learning results of iris flower classification problem – two hidden neurons

Sim. No.		1	2	3	4	5	6	7	8	9	10	Mean
No. of	PIL	622	650	675	597	413	606	610	597	413	606	578.9
epohs	BP	30214	35746	35727	37104	35531	35975	38205	37104	25531	35975	34711

7.6 Case 6: Nonlinear control of magnetic levitation using Virtual Controller approach

Magnetic Levitation (MagLev) systems have practical importance in many engineering systems such as high-speed maglev passenger trains, frictionless bearings, levitation of wind tunnel models, levitation of metal slabs during manufacturing, etc.. Maglev systems are usually open-loop unstable and are highly nonlinear which presents a challenge in designing the feedback controllers.

Figure 28 depicts a typical schematic diagram of a magnetic levitation system which consists of a ferromagnetic ball suspended in a voltage-controlled magnetic field. Only the vertical motion is considered. The nonlinear model of the MagLev system is described by the following nonlinear differential equations:

$$\frac{dp}{dt} = v \tag{111}$$

$$Ri + \frac{d(L(p)i)}{dt} = e \tag{112}$$

$$m\frac{dv}{dt} = mg - C\left(\frac{i}{p}\right)^2 \tag{113}$$

where p denotes the ball's position, v is the ball's velocity, i is the current in the coil of the electromagnet, e is the applied voltage (control variable of the MagLev system), R is the coil's resistance, L is the coil's

inductance, g is the gravitational constant (=9.81m/s^2), C is the magnetic force constant, and m is the mass of the levitated ball. The inductance L is a nonlinear function of the ball's position p, and can be approximated by

$$L(p) = L_0 + \frac{2c}{p}, \qquad p > 0 \tag{114}$$

The parameters of this magnetic levitation system are as follows [26]: R = 28.7 Ω, L_0 = 0.65 H, C = 1.24×10^{-4} Nm^{2}A^2, and m = 11.87 g.

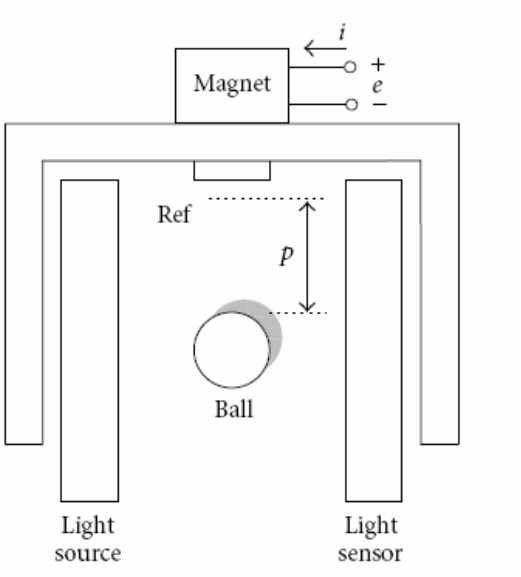

Figure 28 Schematic diagram of a magnetic levitation system

The objective is to design a controller which keeps the position of the ball tracking a reference level. A novel controller design method, which can be named as Virtual Controller (VC) approach, is used. Since the idea of VC is relatively new, a brief introduction is given below.

Consider a data-based output-tracking control problem: Given a set of input-output data of a (possibly nonlinear) system which contains "rich-enough" information about the character of the system, design a

(nonlinear) feedback controller such that the controlled system output can track a reference signal. The conventional approach of designing the controller usually consists of two phases: first identify a suitable model of the system based on the input-output data, and then perform the controller design based on the identified model. The main reason of using this "indirect" strategy is that most of the developed controller design methods are essentially model-based.

An appealing alternative to the conventional strategy is a "direct" strategy which approaches the controller design task *directly* from the input-output data without resorting to the intermediate step of a system model. The fundamental idea of this strategy is fact that all the information contained in the model about the system comes essentially from the input-output data. The Virtual Controller approach is a recently proposed data-based (or so-called *model-free*) controller design method. The VC method is particularly appealing for those complicated nonlinear system control designs where it is very hard (if not impossible) to find a suitable systematic controller design method. The fundamental idea of VC method is illustrated in Figure 29. The VC method is characterized by a "virtual closed-loop" featuring three ingredients:

1) Virtual Closed-Loop (VCL) - the input/output data (u, y) are assumed to be the results of a virtual closed-loop control - which actually does not exist but is imaginary;
2) Virtual Reference (VR) – The VCL is assumed to be driven by a virtual reference signal - again which does not actually exist but is virtual;
3) Virtual Reference Model (VRM) – Under the VCL, the closed-loop performance, which is characterized by the dynamic response from the virtual reference to the controlled output, is considered to be a perfect one - represented by a *reference model* (*i.e.*, a virtual closed-loop transfer function). Theoretically, the implicit essential assumption is that the VRM belongs to the class of achievable closed-loop transfer function.

The controller design of the VC method basically consists of the following two steps.

a) Since it is assumed that the measured output *y* is the output of a closed-loop control system matching a known reference model exactly, this implies that the output signal y can be regarded as the response of the reference model to a *virtual reference* input, the virtual reference can thus be derived as the response of the inverse reference model to the measured output.

b) Since the input (virtual reference plus the measured system output) and output (the known control input to the system) to the virtual controller are all available, the task of controller design is thus transformed to a pure system identification problem, in which the virtual controller rather than the plant is to be identified.

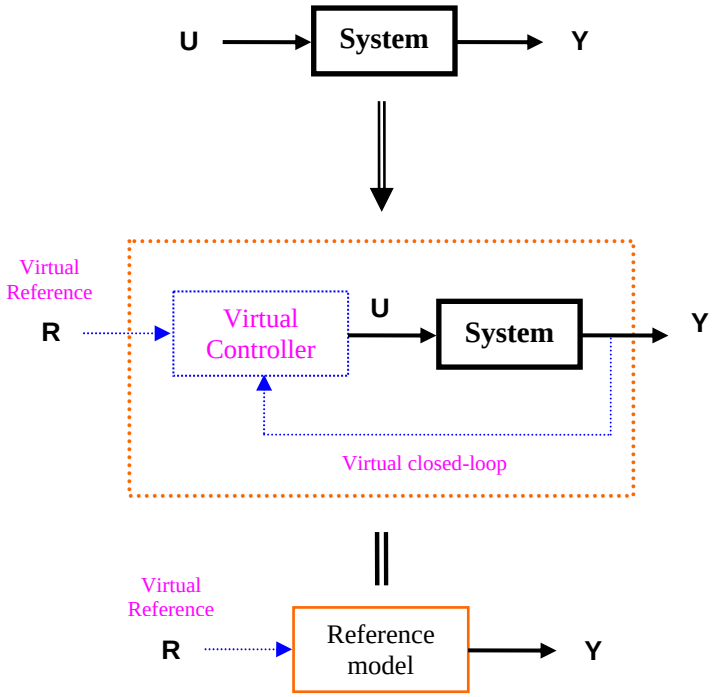

Figure 29 Schematic diagram of Virtual Control

Any standard system identification techniques can be applied to the identification of the virtual controller. Neural networks can be used when the virtual controller to be identified is known to be nonlinear. It is has been found that the key factors to the success of applying the VC method are the determination of a proper reference model and the determination of a proper controller structure.

In this MagLev case, a MLP neural network is used as the nonlinear controller with its parameters to be trained using experimental I/O data from the system. Since the MagLev is open loop unstable, in order to obtain a set of I/O data for the virtual nonlinear controller training, a preliminary linear controller was designed to stabilize the system. Then a sequence of 2 seconds external reference signals was applied to the closed-loop system and the I/O data of the system, as shown in Figure 30, was recorded with the sampling rate of 1000Hz. The reference signal was designed to cover all the possible I/O working range of the system (*i.e.*, to achieve the so-called *sufficient of excitation*). Having done this, the next step was to choose a proper reference model. The reference model was chosen to be:

$$M(z) = \frac{0.0198}{z - 0.9802} \tag{115}$$

It is noted that the selection of the reference model is somewhat heuristic and often involves a trial-and-error process. A bad closed-loop performance of the system is often an indication of poor selection of the reference model rather than an insufficient training of the neural controller. The virtual reference signal can then be obtained by passing the measured system output through the inverse of the reference model.

The next step is to select the structure of the neural controller. This comprises two aspects: the order of the controller and the structure of the neural network. In the case, an MLP with 10 hidden neurons was used to approximate the following 4^{th}-order nonlinear controller:

$$e_k = f_C(e_{k-1}, e_{k-2}, e_{k-3}, e_{k-4}, p_k, p_{k-1}, p_{k-2}, p_{k-3}, p_{k-4}, i_k, i_{k-1}, i_{k-2}, i_{k-3}, i_{k-4}, r_k) \tag{116}$$

where k represents the time instant, r represents the reference level, f_C is the (unknown) virtual controller function to be approximated by neural networks. It is assumed that the current in the coil of the electromagnet is also available for feedback control.

The learning rate was chosen to be 0.0002 for both gradient-descent and PIL algorithms. The learning curves for both algorithms are shown in Figure 31. It can be seen that the PIL algorithm is superior to the standard BP in convergence speed. Figure 32 shows the closed-loop response of the MagLev system to a

reference signal with the neural controller, with the neural weights being trained by PIL algorithm. It is seen that the closed-loop response is very similar to the reference model response (almost indistinguishable from the plots).

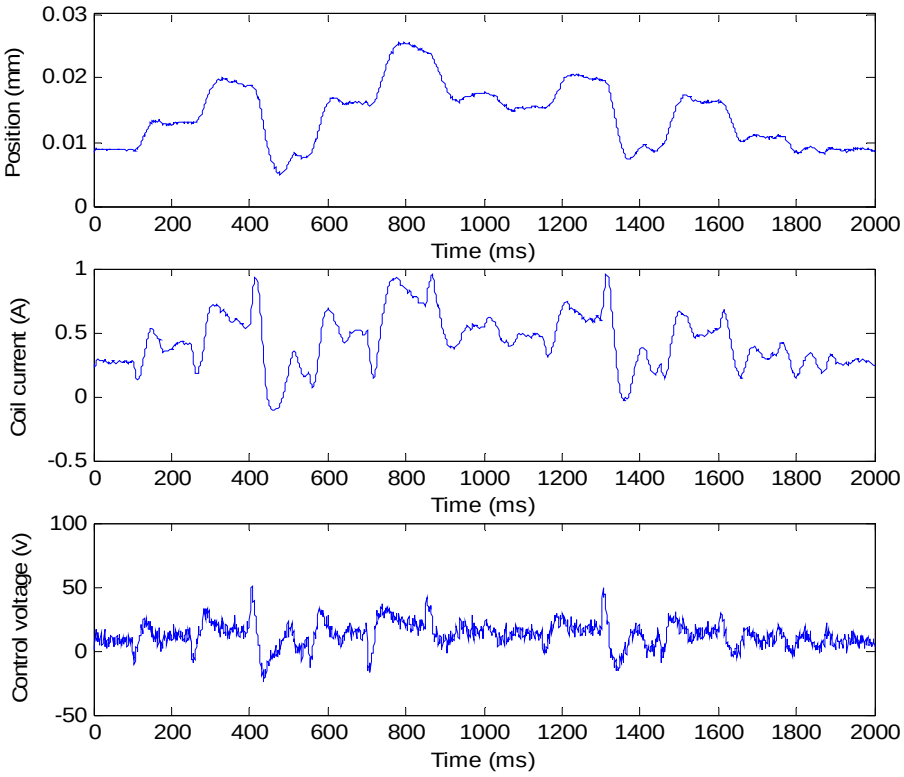

Figure 30 Data of state/control variables used for training the controller

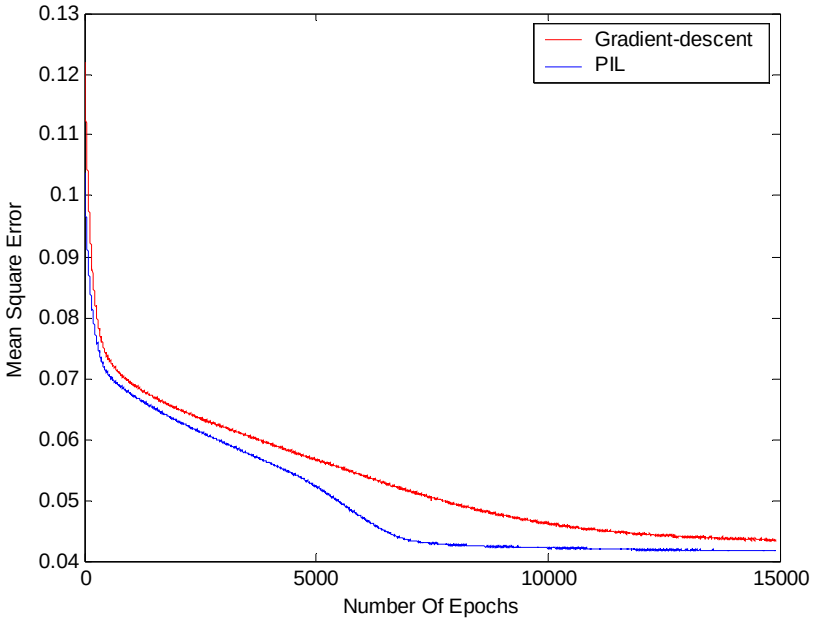

Figure 31 Learning curves for virtual controller

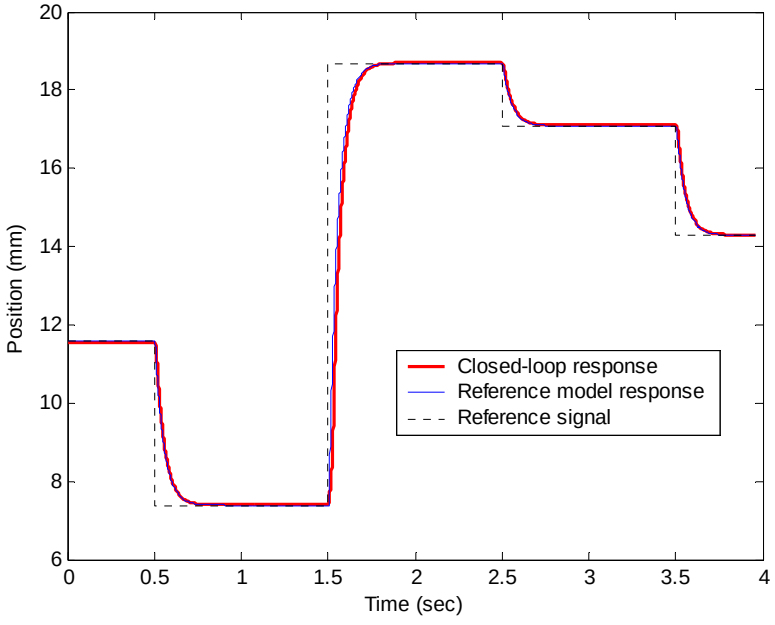

Figure 32 Closed-loop response of the MagLev system with neural controller

7.7 Case 7: Breast cancer diagnosis

Breast cancer diagnosis is a well-known benchmark problem originally obtained from the University of Wisconsin Hospitals, Madison, and is available from the PROBEN1 – a collection of benchmark problems for neural network learning [27]. The task is to classify a tumor as either benign or malignant based on cell descriptions consisting of 9 variables. A total amount of 699 examples are available, from which 525 randomly selected examples are used for neural network training while the remaining 174 examples are left for testing.

Three hidden neurons were used since it was observed that this structure gives the best generalization capability for both PIL and BP networks. The learning rate was set to 0.05 which was found to be well-tuned for both networks. Twenty runs of learning simulation were conducted; the number of training epochs was set to 10,000 in all runs since it was observed that this number of epochs is sufficient for both networks to reach a steady state error. The simulation results in terms of Mean Square Error (MSE) and classification error (both misclassification counts and misclassification percentage) for both the training data and the test data were reported in Table 2. The last row of the table shows the statistics (i.e., average) of the 20 simulated results.

The following conclusions can be drawn from the simulation results.

1) For the training data set:
    - Most of the classification errors, i.e., for 17 out of 20 runs, of the network trained by the PIL algorithm are less than that of the network trained by the standard BP algorithm;
    - For the majority of runs, i.e., 14 out of 20, the MSE's of the network trained by PIL are less than those of the network trained by standard BP;
    - Statistically, the mean classification error of the network trained by the PIL algorithm is about half of that of the network trained by standard BP algorithm (1.05% vs. 1.92%);
    - Statistically, the average MSE of the network trained by the PIL algorithm is slightly less that that of the network trained by the standard BP algorithm (2.97×10^{-2} vs. 3.32×10^{-2});

- These above facts clearly demonstrate that the training performance of the PIL algorithm is significantly better than that of the standard BP algorithm for this classification problem.

Table 2 Learning results of breast cancer diagnosis problem – three hidden neurons

	Training Data Set (data size = 525)						Testing Data Set (data size = 174)					
	PIL			BP			PIL			BP		
Sim. No.	MSE ($X10^{-2}$)	Classif. Err.	Classif. Err.(%)	MSE ($X10^{-2}$)	Classif. Err.	Classif. Err.(%)	MSE ($X10^{-2}$)	Classif. Err.	Classif. Err.(%)	MSE ($X10^{-2}$)	Classif. Err.	Classif. Err.(%)
1	2.23	4	0.76	3.75	14	2.67	4.89	3	1.72	6.23	7	4.02
2	3.16	6	1.14	1.9	3	0.57	3.57	3	1.72	11.13	7	4.02
3	4.16	8	1.52	3.34	10	1.9	6.13	4	2.3	5.25	7	4.02
4	2.2	4	0.76	3.3	9	1.71	6.29	4	2.3	4.82	6	3.45
5	2.14	4	0.76	3.77	11	2.1	9.03	6	3.45	6.14	7	4.02
6	2.2	4	0.76	3.77	13	2.48	6.24	4	2.3	6.22	7	4.02
7	3.67	7	1.33	3.74	13	2.48	3.46	2	1.15	5.76	6	3.45
8	3.65	7	1.33	3.32	10	1.9	4.8	3	1.72	4.71	7	4.02
9	2.73	5	0.95	3.75	13	2.48	5.55	4	2.3	5.85	6	3.45
10	3.69	7	1.33	3.77	13	2.48	4.84	3	1.72	5.9	6	3.45
11	2.22	4	0.76	3.76	12	2.29	4.86	3	1.72	6.18	7	4.02
12	3.69	7	1.33	3.09	8	1.52	4.88	3	1.72	11.05	12	6.9
13	2.2	4	0.76	1.88	3	0.57	5.54	4	2.3	11.94	8	4.6
14	3.71	7	1.33	3.77	13	2.48	4.61	3	1.72	6.13	7	4.02
15	3.65	7	1.33	1.88	3	0.57	4.27	3	1.72	12.49	8	4.6
16	3.34	5	0.95	3.5	6	1.14	6.2	4	2.3	8.84	8	4.6
17	3.68	7	1.33	3.77	13	2.48	5.74	4	2.3	5.97	7	4.02
18	1.68	3	0.57	3.76	13	2.48	7.63	5	2.87	5.82	6	3.45
19	2.21	4	0.76	3.28	12	2.29	5.31	4	2.3	5.21	7	4.02
20	3.16	6	1.14	3.34	9	1.71	5.5	4	2.3	5.19	7	4.02
Mean	2.9685	5.5	1.045	3.322	10.05	1.915	5.467	3.65	2.0965	7.0415	7.15	4.1085

2) For the test data set:

- The classification errors of the network trained by the PIL algorithm are all less than those of the network trained by the standard BP algorithm in all runs;

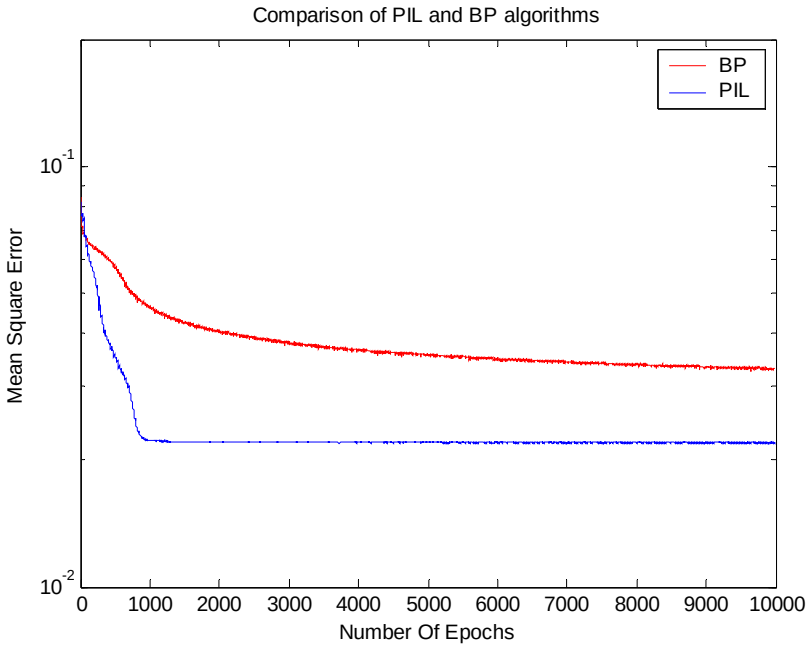

Figure 33 A typical learning curves for Breast Cancer Diagnosis - MSE

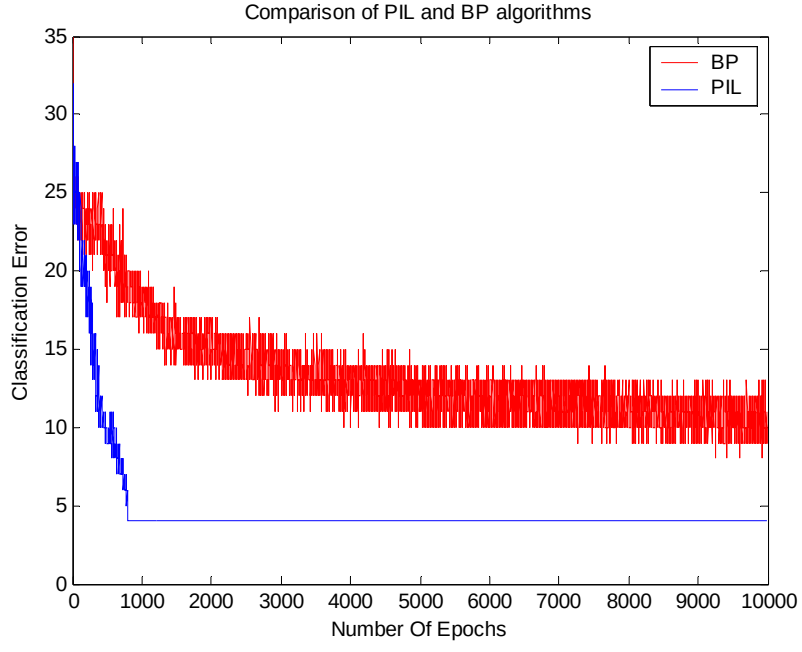

Figure 34 A typical learning curves for Breast Cancer Diagnosis – Classification Error

- For majority of runs, i.e., 12 out of 20, the MSE's of the network trained by the PIL are less than that of the network trained by the standard BP;
- Statistically, the mean classification error of the network trained by the PIL algorithm is about half of that of the network trained by the standard BP algorithm (2.09% vs. 4.11%);
- Statistically, the average MSE of the network trained by the PIL algorithm is appreciably less that that of the network trained by the standard BP algorithm (5.47×10^{-2} vs. 7.04×10^{-2});
- These facts clearly demonstrate that the generalization performance of the PIL algorithm is significantly better than the standard BP algorithm for this classification problem.

Figure 33 and Figure 34 show the typical learning curves of MSE *vs*. Number of Epochs and Classification Error *vs*. Number of Epochs, respectively. It is seen that the PIL algorithm is superior to the standard BP algorithm in terms of convergence speed and the final values as well for this classification task.

7.8 Case 8: Modeling of the NOx emissions of a diesel engine

The task is to build up a model which is capable of predicting the emission of the oxides of nitrogen (or, NOx) of a diesel engine, using the measured engine speed and torque. Two sets of data were collected at the Engine & Emissions Research Laboratory at West Virginia University, from two cycles of engine dynamometer tests, respectively. The data were sampled at 1 Hz. One set of data, consisting of 900 data samples shown in Figure 35, was used for training the model; the other data set, consisting of 1400 data samples shown in Figure 36, was used for testing the quality of the trained model.

The following first-order model was used:

$$\widehat{NOx}_k = f_{NN}\left(\widehat{NOx}_{k-1}, T_k, S_k\right) \tag{117}$$

where $\widehat{NOx}_k$, T_k, S_k are the estimation of NOx, engine torque and speed at time instant *k*, respectively.

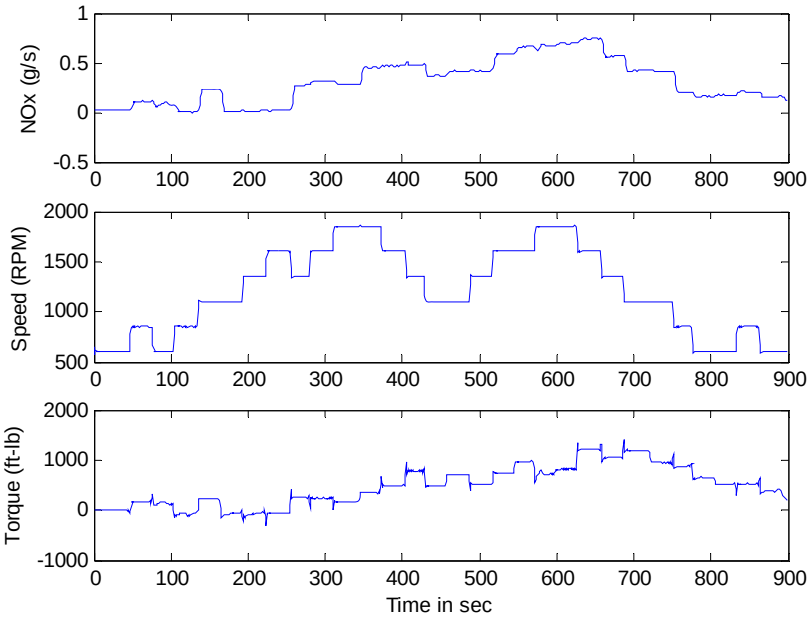

Figure 35 Engine emission dynamometer test data used for training

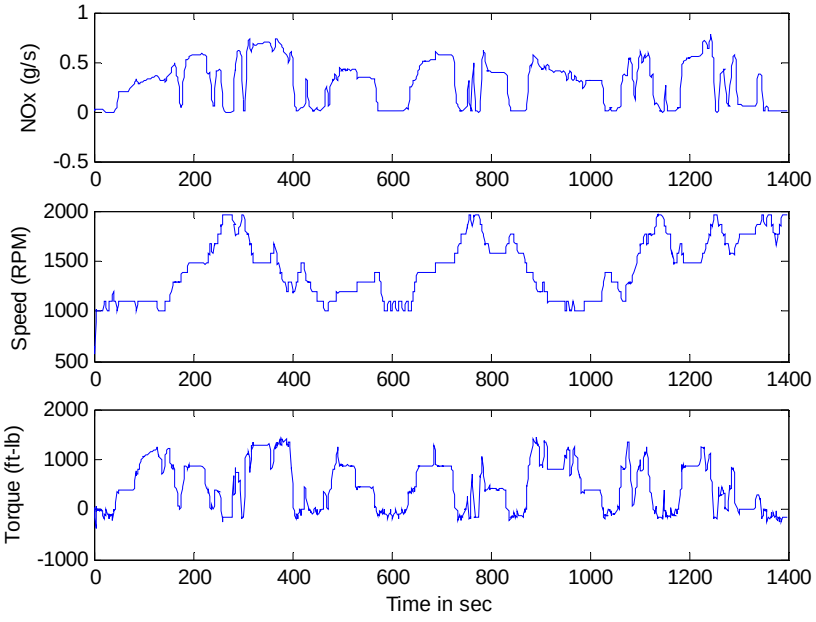

Figure 36 Engine emission dynamometer test data used for testing

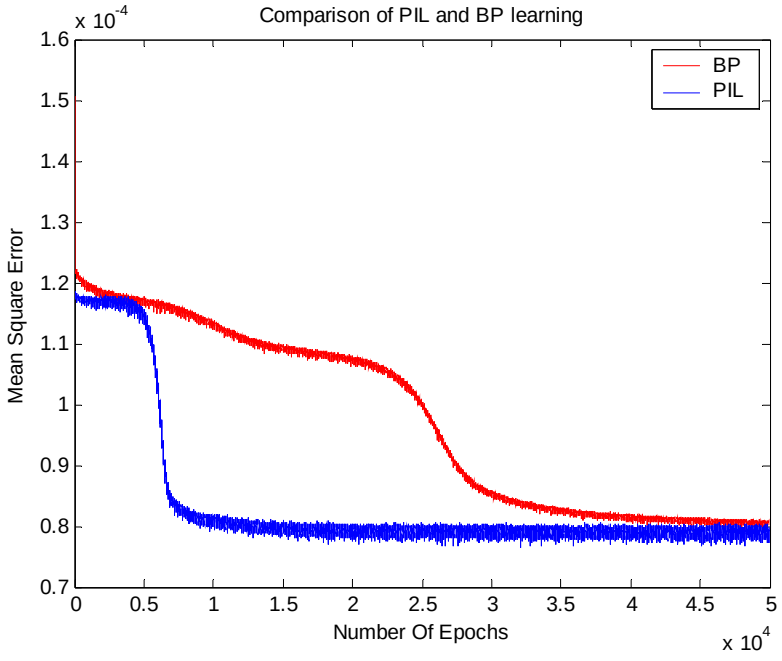

Figure 37 Learning curves for NOx emission model

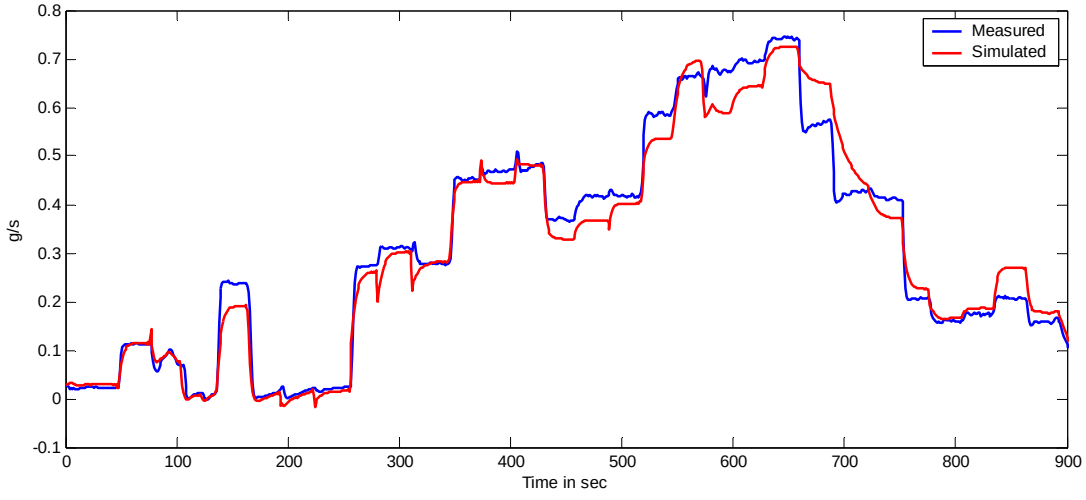

Figure 38 Simulation of the NOx emission model on training data

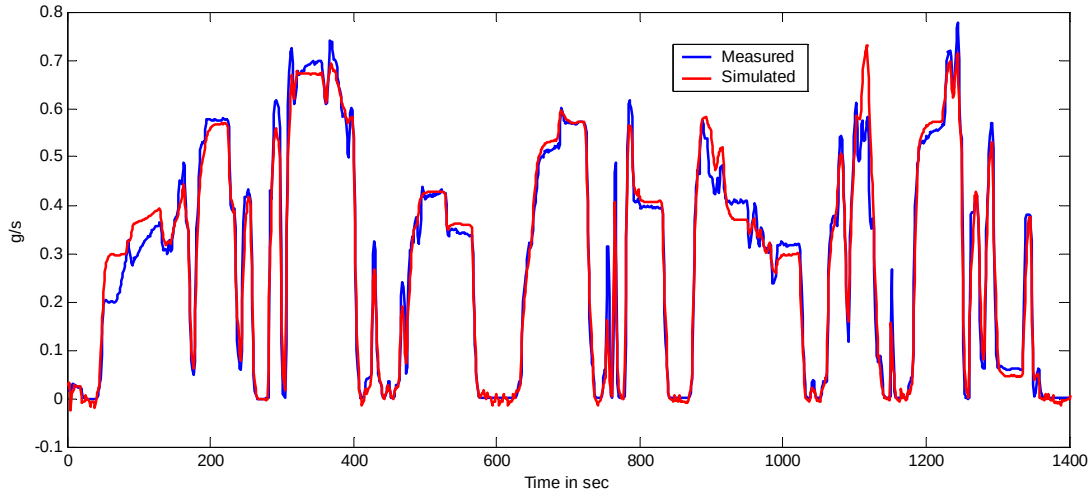

Figure 39 Simulation of the NOx emission model on testing data

Four hidden neurons were used for the model, and the learning curves are shown in Figure 37. It is seen that the PIL algorithm converges remarkably faster than the standard BP algorithm. Figure 38 and Figure 39 show the measured NOx vs. the simulated NOx of the model trained by the PIL algorithm, on both training data and testing data, respectively.

Chapter 8 Numerical experiments – comparison with the SDLM algorithm

8.1 Brief introduction of the Stochastic Diagonal Levenberg-Marquardt method

The Stochastic Diagonal Levenberg-Marquardt (SDLM) method, which was developed in [1], is a well-known simplified 2nd-order stochastic learning algorithm for feedforward neural networks. This algorithm is obtained by modifying the well-known Levenberg-Marquardt optimization algorithm in three ways: First, it applies the stochastic learning principle to the standard Levenberg-Marquardt algorithm by updating the network parameters via the Levenberg-Marquardt algorithm once every single training pattern is presented; Second, it uses only the diagonal terms of the approximated Hessian matrix, thus significantly reduces the computational cost; Third, it uses a first-order filter to smooth out the noisy (due to the stochastic nature of the algorithm) diagonal terms of the approximated Hessian matrix. The SDLM is an efficient learning algorithm which usually outperforms the standard BP algorithm. More importantly, to the best knowledge of the author, the SDLM is perhaps the simplest stochastic "second-order" algorithm since its computational cost is the same order as the standard BP algorithm, i.e., it requires only $O(n)$ operations in each iteration. The SDLM algorithm can be briefly described as follows.

Given the square error for the current training pattern:

$$E_p = \frac{1}{2}\left(\hat{y}(\mathbf{\theta}) - y\right)^2, \quad \mathbf{\theta} = [\theta_1 \quad \theta_2 \quad \cdots \quad \theta_M] \tag{118}$$

where the $\mathbf{\theta}$ is the weight vector of the neural network, and $\hat{y}(\mathbf{\theta})$ is the estimated target value with given input and neural weights. The first and second derivatives of E_p are given by:

$$\frac{\partial E_p}{\partial \theta_i} = \left(\hat{y}(\mathbf{\theta}) - y\right)\frac{\partial \hat{y}(\mathbf{\theta})}{\partial \theta_i} \tag{119}$$

$$\frac{\partial^2 E_p}{\partial \theta_i^2} = \left(\frac{\partial \hat{y}(\mathbf{\theta})}{\partial \theta_i}\right)^2 + \left(\hat{y}(\mathbf{\theta}) - y\right)\frac{\partial^2 \hat{y}(\mathbf{\theta})}{\partial \theta_i^2} \tag{120}$$

respectively. For Levenberg-Marquardt method, it is assumed that

$$\frac{\partial^2 \hat{y}(\mathbf{\theta})}{\partial \theta_i^2} \approx 0 \tag{121}$$

so,

$$\frac{\partial^2 E_p}{\partial \theta_i^2} \approx \left(\frac{\partial \hat{y}(\mathbf{\theta})}{\partial \theta_i} \right)^2 \tag{122}$$

The running estimate of the diagonal second derivative, denoted by $\left\langle \frac{\partial^2 E_p}{\partial \theta_i^2} \right\rangle$, is given by:

$$\left\langle \frac{\partial^2 E_p}{\partial \theta_i^2} \right\rangle_{new} = (1-\gamma) \left\langle \frac{\partial^2 E_p}{\partial \theta_i^2} \right\rangle_{old} + \gamma \frac{\partial^2 E_p}{\partial \theta_i^2}, \qquad 0 < \gamma < 1 \tag{123}$$

It is worth noting that this is equivalent to passing the stochastic second derivative $\frac{\partial^2 E_p}{\partial \theta_i^2}$ through a low-pass filter $\frac{\gamma}{1-(1-\gamma)z^{-1}}$ to obtain a running estimate of the diagonal second derivative.

Instead of using a constant learning rate in standard BP, the learning rate in this algorithm is given by:

$$\eta_i = \frac{\varepsilon}{\left\langle \frac{\partial^2 E_p}{\partial \theta_i^2} \right\rangle + \mu} \tag{124}$$

where ε is the (global) learning rate, and μ is a small parameter to prevent η_i from blowing up in case the second derivative is close to zero. The parameter update rule is given by

$$\Delta \theta_i = \eta_i \frac{\partial E_p}{\partial \theta_i} = \frac{\varepsilon}{\left\langle \frac{\partial^2 E_p}{\partial \theta_i^2} \right\rangle + \mu} \left(\hat{y}(\mathbf{\theta}) - y \right) \frac{\partial \hat{y}(\mathbf{\theta})}{\partial \theta_i}$$

$$= -\frac{\varepsilon}{\left\langle \frac{\partial^2 E_p}{\partial \theta_i^2} \right\rangle + \mu} \frac{\partial \hat{y}(\boldsymbol{\theta})}{\partial \theta_i} \Delta y, \qquad \Delta y = y - \hat{y}(\boldsymbol{\theta}) \tag{125}$$

It is worth noting that, although the SDLM algorithm is expected to speed up the convergence of the standard BP algorithm, it has some drawbacks. Comparing the SDLM with the standard BP where a single learning rate is to be tuned, there are three parameters to be determined in this algorithm, i.e., the learning rate ε, the filter constant γ and the small constant μ. Also it requires more memory to store the additional states, i.e., the running estimate of the diagonal second derivatives.

In the following two numerical studies, all the relevant conventions/conditions described in the previous chapter are followed.

8.2 Case 1: Peaks function approximation

Again the task is to approximate the MATLAB® Peaks function as expressed by Eq. (108) in section 7.1 of the previous chapter. The objective here is to compare the learning performance of the PIL algorithm with the SDLM algorithm. The BP learning algorithm is also included for better comparison. The training data set is exactly the same as that used in 7.1. The learning rate is set to 0.003 for all the three algorithms which is found to be well-tuned. The other two parameters for the SDLM algorithm are tuned to $\mu = 0.01$ and $\gamma = 0.00001$, respectively, which can give satisfactory performance. Five numerical simulation are conducted, with each having different number of hidden neurons from 10 to 30. The training epochs are sufficient for each of the learning algorithms to reach steady state MSE values. Figure 40 through Figure 44 show the learning curves of the five numerical simulations. It is seen that the PIL algorithm and the SDLM algorithm have a similar convergence speed in reaching their own steady state error values. However, it is noted that only for the case of 20 hidden neurons (corresponding to Figure 42) the SDLM algorithm

converges to a steady state error value which is smaller than that achieved by the PIL algorithm, otherwise the PIL algorithm converges to smaller values of steady state error than that achieved by SDLM algorithm, clearly indicating that the PIL algorithm have a much higher chance (4 out of 5) of achieving a better solution.

It is worth pointing out that from the algorithm formulae it can be easily seen that, while the computational cost of the PIL algorithm is almost the same as that of the standard BP algorithm, the SDLM needs slightly more computation time mainly due to the additional computational cost of estimating the filtered diagonal Hessian. The following numerical experiment result supports the claim.

For the Peaks function approximation problem discussed in this section, set the number of hidden neurons to 20. Run each of the three algorithms, BP, PIL and SDLM, on Pentium IV computer with 2.8 GHz CPU for 10,000 epochs, respectively. Then the computation time obtained for each of the runs is listed below:

Table 3 Computation time for BP, PIL and SDLM algorithms

Algorithm	BP	PIL	SDLM
Computation Time	23’57”	24’9”	25’4”
Relative computation time	100%	100.82%	104.66%

It is straightforward to calculate from the data in Table 3 that, comparing with the standard BP algorithm, the PIL algorithm takes less than one percent (i.e., 0.82%) more computation time than that of the standard BP algorithm, practically implies nearly the same computational cost; while the SDLM takes slightly more computation time (i.e., 4.66%) than that of the standard BP.

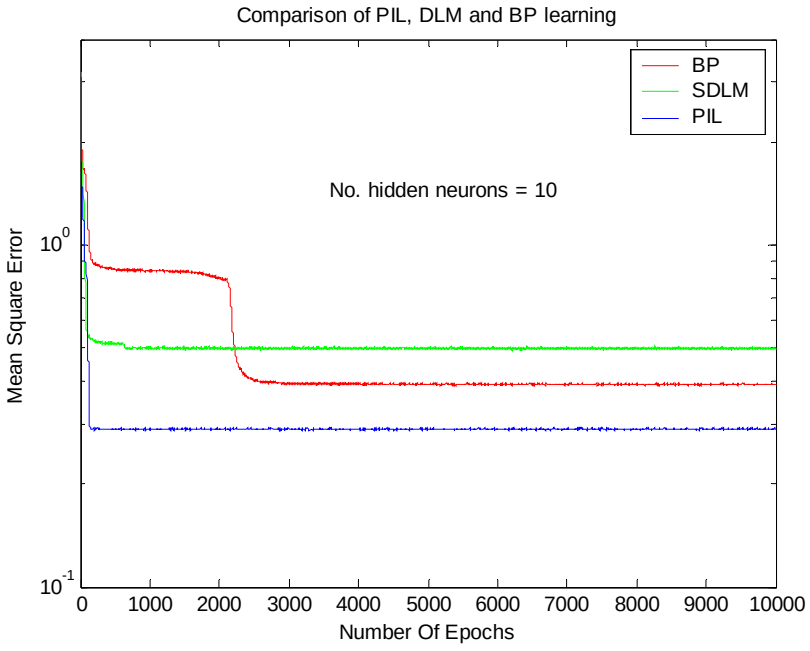

Figure 40 Learning curves for Peaks function – 10 hidden neurons

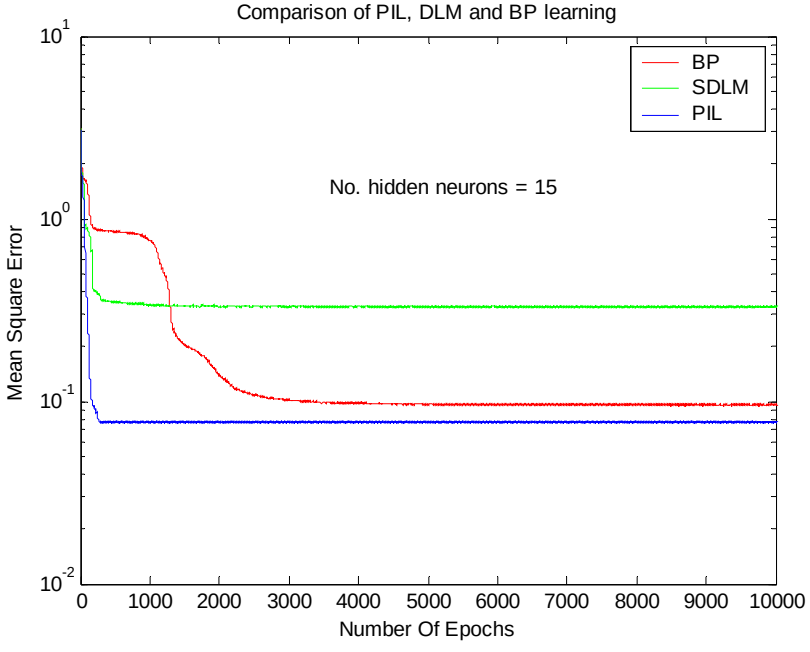

Figure 41 Learning curves for Peaks function – 15 hidden neurons

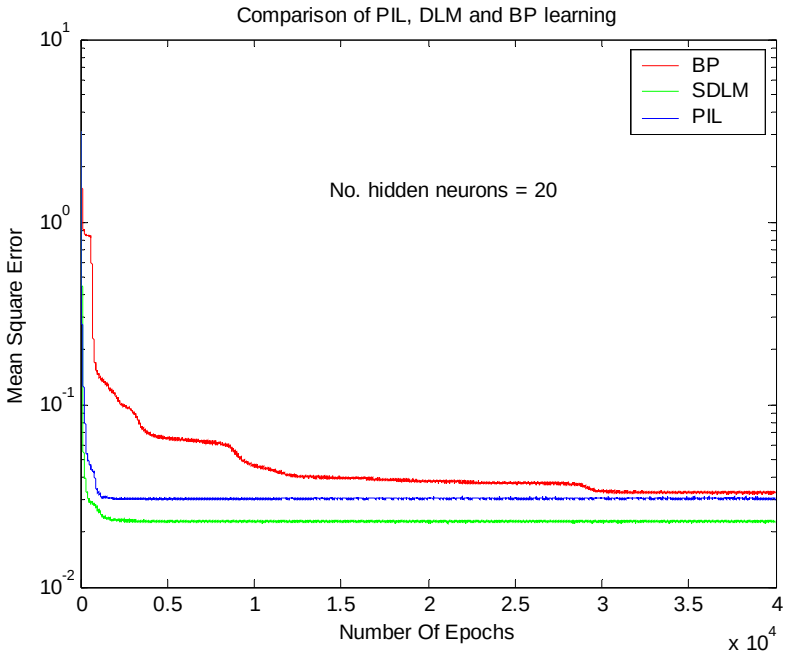

Figure 42 Learning curves for Peaks function – 20 hidden neurons

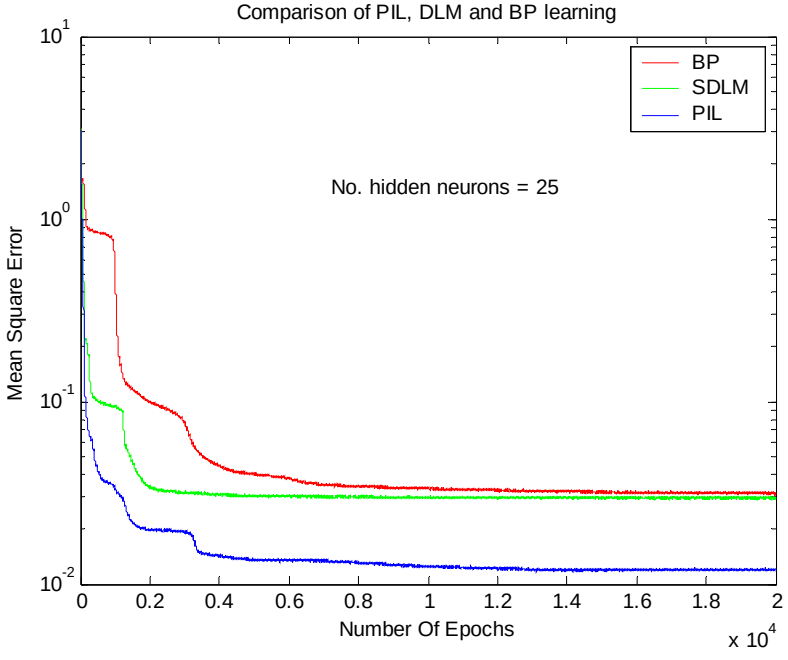

Figure 43 Learning curves for Peaks function – 25 hidden neurons

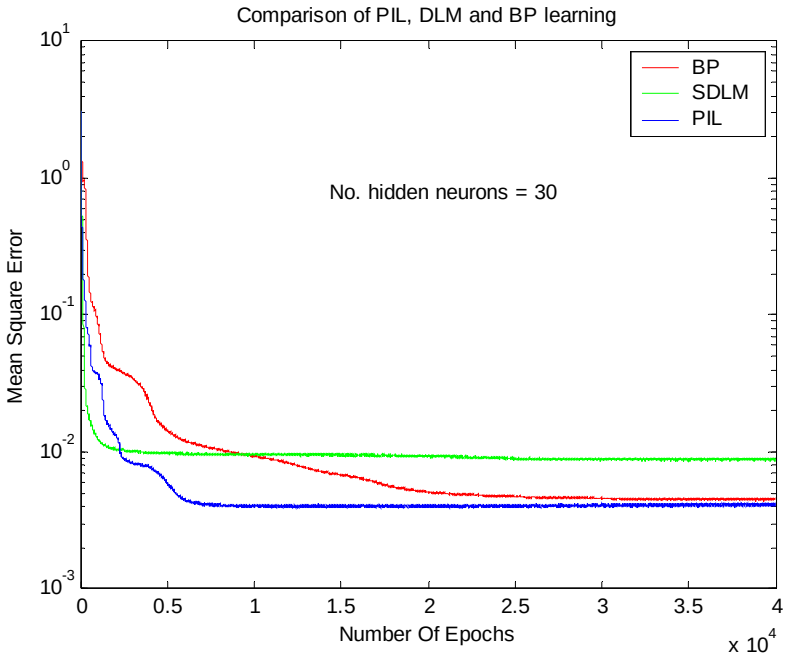

Figure 44 Learning curves for Peaks function – 30 hidden neurons

8.3 Case 2: Gabor function approximation

As in section 7.2 of the previous chapter, the task is to approximate the 2-D Gabor function as expressed by Eq. (109). The objective here is to compare the learning performance of the PIL algorithm with the SDLM algorithm. Note that the results of BP learning algorithm are NOT included for this case study since it is observed that the BP algorithm never starts to learn in all the numerical simulations in this section, as is the case shown in Figure 17 and Figure 18 of the previous chapter. The training data set is exactly the same as that used in 7.2. The learning rate is set to 0.001 for all the three algorithms which is found to be well-tuned. The other two parameters for the SDLM algorithm are tuned to $\mu = 0.01$ and $\gamma = 0.00001$, respectively, which can give satisfactory performance. Five numerical simulations are conducted, with each having a different number of hidden neurons from 10 to 30. The training epochs are sufficient for each of the learning algorithms to reach steady state MSE values. Figure 45 through Figure 49 show the learning curves

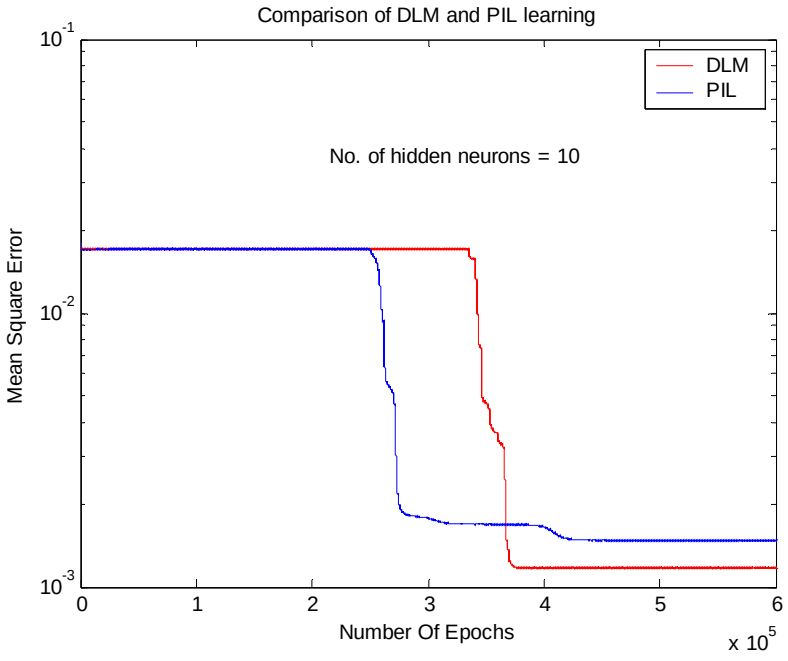

Figure 45 Learning curves for 2-D Gabor function – 10 hidden neurons

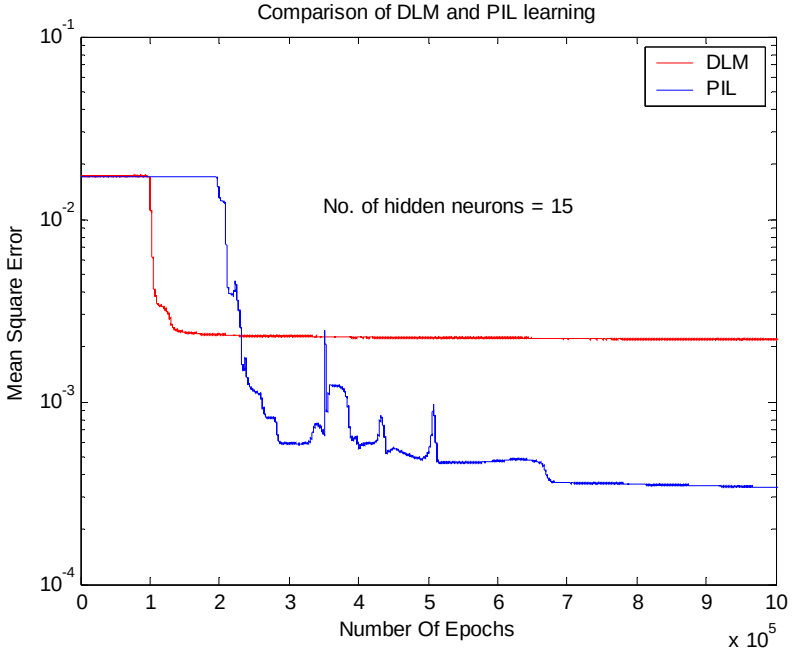

Figure 46 Learning curves for 2-D Gabor function – 15 hidden neurons

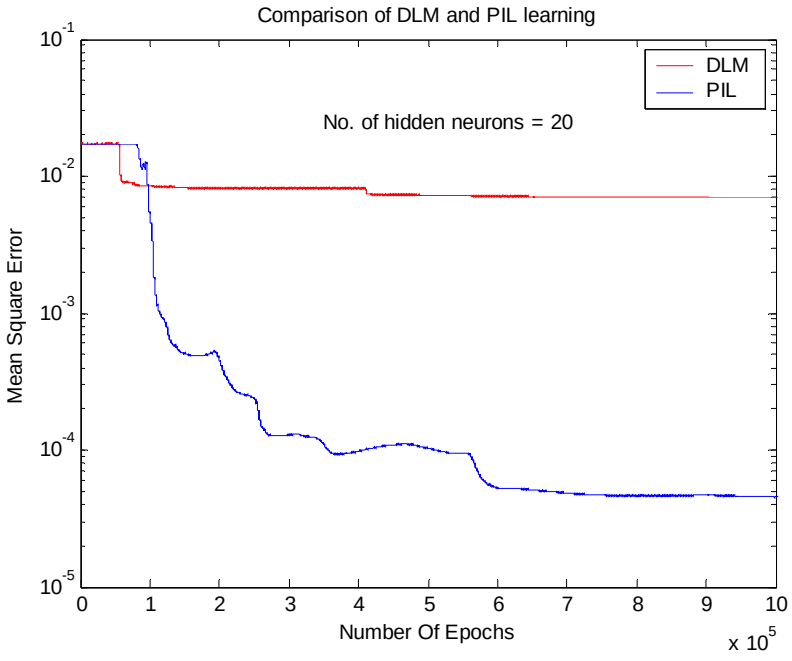

Figure 47 Learning curves for 2-D Gabor function – 20 hidden neurons

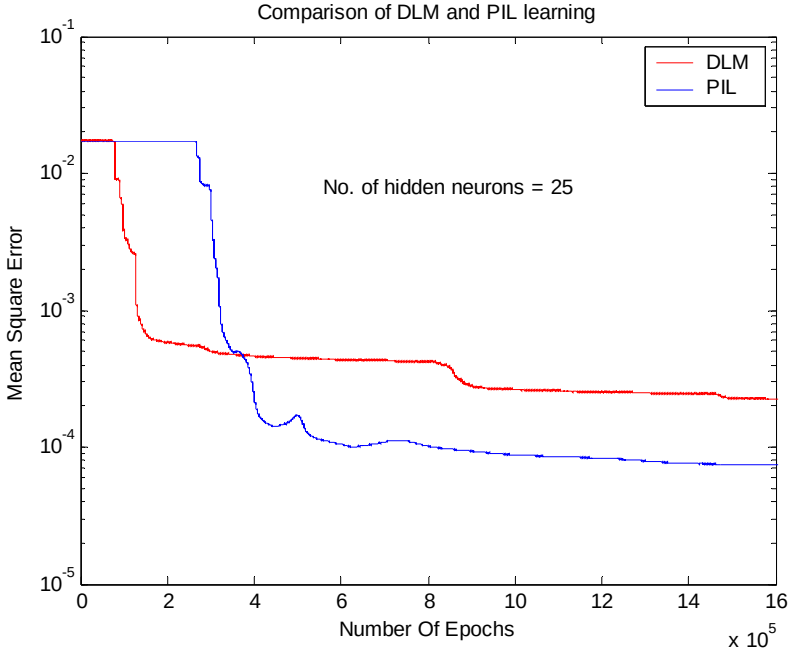

Figure 48 Learning curves for 2-D Gabor function – 25 hidden neurons

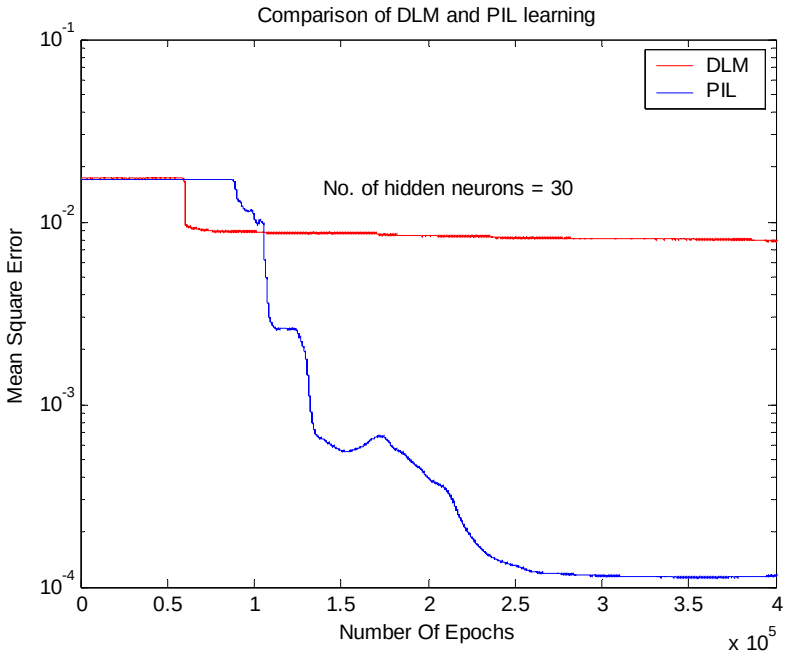

Figure 49 Learning curves for 2-D Gabor function – 30 hidden neurons

of the five numerical simulations. It is seen that for 4 out of 5 runs (see Figure 46 through Figure 49), the PIL algorithm converges to appreciably smaller MSE values than the SDLM algorithm does, indicating that the PIL algorithm has a higher chance of achieving a better solution than the SDLM algorithm does.

Chapter 9 Conclusions

The contribution of this work to the neural networks community is the proposition, development, analysis and numerical validation of a novel stochastic learning algorithm for feedforward neural networks: the Parameter Incremental Learning (PIL) algorithm.

The essence of the PIL strategy is that in the process of adapting the network to training data by adjusting its parameters, the deformation of the network is also explicitly required to be preserved to certain extent. A general PIL problem at the global level (or network level) is initially proposed. By using the first-order approximating technique, the analytical formula is derived. However, the general PIL algorithm at the global level is generally not realizable since it is usually not possible to find the analytical solution to the integral of the outer product of the partial derivative of the network function with respect to the parameter vector. And even if the integral can be obtained, the computational cost associated with the inverse of a square matrix, which has a dimension as high as the number of adjustable network parameters, is enormous.

As an attempt to deal with the above-mentioned difficulty, the general PIL problem at the global level is modified in such a way that the performance of preservation is against the deformation of each individual neuron instead of the network as a whole. For this sub-optimal strategy, the analytical solution is presented, with the “free parameters” being determined by normalizing the error-reduction property of the algorithm to that of the standard gradient descent algorithm. An alternative interpretation of the PIL algorithm at the neuron level, which reveals some insights of the algorithm, is also given, in that the problem is posed as if only a single neuron in the network were to be adapted when a training pattern is presented. And the performance index which bears the essence of incremental learning is posed as to minimize the deformation of this single neuron subject to a certain amount of error reduction (i.e., adaptation to the training pattern) specified by an equality constraint. Though started from a different perspective, the resulting formulae remain essentially the same.

The PIL algorithm for MLP is then derived, based on the general PIL algorithm at the neuron level and

the introduction of an extra fictitious input to the neuron. The derived PIL algorithm for MLP is as simple as the conventional (on-line) Back-Propagation algorithm so it can be used in all situations wherever the standard stochastic BP method is applicable.

To gain further insights of the new algorithm, an analytical comparison of the PIL algorithm with another recently developed, principled algorithm called Natural Gradient Descent algorithm, is made. It is shown that, although they stem from different theoretical perspectives, the NGD algorithm for the Gaussian stochastic feedforward neural network model with uniform distribution of input variable, and the General PIL algorithm, are essentially the same up to a selection of a free parameter. To further reveal the intrinsic relationship between the two algorithms, another comparison is made between the parameter update laws of the two algorithms for the simplest multi-input, single output MLP, i.e., the MLP with single non-linear neuron. It is shown that the two algorithms share a strong similarity which distinguishes them from the standard BP algorithm.

Extensive numerical studies have been carried out. The problems used for the simulation include function approximation, classification, dynamic system modeling and neural controller. The following conclusions can be drawn from the numerical experiments.

1). The PIL algorithm is remarkably superior to the standard on-line BP learning algorithm in convergence speed;

2). Compared to the standard BP algorithm, the PIL algorithm has a much better chance to get rid of the plateau area, which is a frequently encountered problem in standard BP algorithm;

3). The learning rate that is well-tuned for the standard BP algorithm is also well-tuned for the PIL algorithm, and vice versa. This implies that the PIL can readily replace the standard on-line BP algorithm already in use to get better performance even without re-tuning the learning rate;

4). The computation cost per epoch for PIL algorithm and the standard BP algorithm are almost the same;

5). Compared to the SDLM algorithm, although the convergence speeds are similar, the PIL algorithm has a significantly better chance to find a better solution;

6). Compared to the SDLM algorithm, the PIL algorithm is easier to use since the it has a single parameter (i.e., the learning rate) to be tuned, while the SDLM algorithm has three parameters to be tuned simultaneously;

7). The SDLM algorithm requires more memory to store the parameters (approximately double) than the PIL algorithm does, this could be a concern for those applications where small memory requirement is desired.

In general, the PIL algorithm is shown to have the potential to replace the standard on-line BP algorithm as a standard stochastic (or on-line) learning algorithm for MLP. Particularly, the PIL algorithm is a strong candidate to solve difficult problems where the conventional BP is unable to find a satisfactory solution.

References

[1]. Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klause-Robert Müller, "Efficient BackProp," In Genevieve B. Orr and Klause-Robert Müller, editors, "Neural networks: tricks of the trade", pp. 9-53, Springer-Verlag, Berlin Heidelberg, 1998.

[2]. Russell D. Reed and Robert J. Marks II, "Neural smithing: supervised learning in feedforward artificial neural networks," MIT Press, Cambridge, 1999.

[3]. Martin Riedmiller, "Advanced supervised learning in multi-layer perceptrons – from backpropagation to adaptive learning algorithms," *International Journal of Computer Standards and Interfaces*, special Issue on Neural Networks, 16(3): 265-278, 1994.

[4]. Roberto Battiti, "First- and Second-Order Methods for Learning: between Steepest Descent and Newton's Method," *Neural Computation*, Vol. 4 No. 2, pp. 141-166, 1992.

[5]. John Platt, "A Resource-Allocating Network for Function Interpolation," *Neural Computation*, Vol. 3 No. 2, pp. 213-225, 1991.

[6]. Simon Haykin, "*Neural Networks: A Comprehensive Foundation*," Prentice Hall, Upper Saddle River, NJ, 1999.

[7]. K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, No. 5, pp. 359–366, 1989.

[8]. K. Hornik, M. Stinchcombe, and H. White, "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks," *Neural Networks*, vol. 3, No. 5, pp. 551–560, 1990.

[9]. D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning internal representation by error propagation," In *Parallel Distributed Processing*, Nature 323, pp. 533-536, 1986.

[10]. D. B. Parker, "Learning logic: Casting the cortex of human brains in silicon," Technical Report 47, Center for Computational Research in Economics and Management Science, MIT, Cambridge, MA 1985.

[11]. P. J. Werbos, "Beyond regression: new tools for prediction and analysis in behavioral sciences," Ph.D.

Thesis, Harvard University, Cambridge, MA 1974.

[12]. A. E. Bryson and Y. C. Ho, “Applied optimal control,” New York: Blaisdell, 1969.

[13]. David Saad, Sara A. Solla, “On-line learning in soft committee machines,” *Physics Review E*, Vol. 52, pp. 4225-4243, 1995.

[14]. D. Plaut, S. Nowlan and G. Hinton, “Experiments on learning by Back-Propagation,” Technical Report CMU-CS-86-126, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1986.

[15]. S. E. Fahlman, “An empirical study of learning speed in back-propagation networks,” Technical Report CMU-CS-88-162, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1988.

[16]. M. Riedmiller and H. Braun, “A directive adaptive method for faster backpropagation learning: the RPROP algorithm,” In H. Ruspini, editor, *Proceedings of the IEEE International Conference on Neural networks (ICNN)*, pp. 586-591, San Francisco, 1993.

[17]. E. M. Johansson, F. U. Dowla and D. M. Goodman, “Backpropagation learning for multi-layer feed-forward neural networks using the conjugate method,” Livermore National Laboratory, Preprint UCRL-JC-104850, 1990.

[18]. Shun-ichi Amari, “Natural Gradient Works Efficiently in Learning,” *Neural Computation*, Vol. 10, No. 2, pp. 251 – 276, 1998.

[19]. H. H. Yang and S. Amari, “Complexity issues in natural gradient descent method for training multilayer perceptrons,” *Neural Computation*, Vol. 10, No. 8, pp. 2137 – 2157, 1998.

[20]. S. Amari, H. Park and K. Fukumizu, “Adaptive method of realizing natural gradient learning for multilayer perceptrons,” *Neural Computation*, Vol. 12, No. 6, pp. 1399 – 1409, 2000.

[21]. Kevin J. Lang and Michael J, Witbrock, “Learning to Tell Two Spirals Apart”, in Proceedings of the 1988 Connectionist Models Summer School, Morgan Kaufmann, 1988.

[22]. Two-spirals classification data set, available at: http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/cmu/

[23]. Schaal, S. and Atkeson, C. G., "Constructive incremental learning from only local information," *Neural Computation*, 10, 8, pp. 2047-2084, 1998.

[24]. R. A. Fisher, "The use of multiple measurements in taxonomic problem," *Annals of Eugenics*, Vol. 7, pp.179–188, 1936.

[25]. Iris classification data set, available at: ftp://ftp.ics.uci.edu/pub/machine-learning-databases/iris/.

[26]. W. Barie and J. Chiasson, "Linear and nonlinear state-space controllers for magnetic levitation," *International Journal of Systems Science*, Vol. 27, No. 11, pp.1153–1163, 1996.

[27]. Lutz Prechelt, "PROBEN1 – A Set of Neural Network Problems and Benchmarking Rules", Technical Report 21/94, University of Karlsruhe, Germany, 1994. Available at ftp://ftp.ira.uka.de/pub/uni-karlsruhe/papers/techreports/1994/1994-21.ps.gz. Data available at ftp://ftp.ira.uka.de/pub/neuron/proben1.tar.gz

Appendix A: Derivation of PIL algorithm for MLP

Note that by first integrating over the fictitious variable v the equation (87) can be written as

$$\tilde{\boldsymbol{\Theta}}(\boldsymbol{\theta}, \lambda) = \int_{\mathbf{u}\in\mathfrak{D}} \boldsymbol{\Theta}_v(\mathbf{u}, \boldsymbol{\theta}, \lambda) d\mathbf{u} \tag{126}$$

where:

$$\boldsymbol{\Theta}_v(\mathbf{u}, \boldsymbol{\theta}, \lambda) \triangleq \int_{-\infty}^{+\infty} (g'(\tilde{s}))^2 \begin{bmatrix} 1 \\ \tilde{\mathbf{u}} \end{bmatrix} \begin{bmatrix} 1 & \tilde{\mathbf{u}}^{\mathbf{T}} \end{bmatrix} dv \tag{127}$$

And (127) can further be written as

$$\boldsymbol{\Theta}_v(\mathbf{u}, \boldsymbol{\theta}, \lambda) = \int_{-\infty}^{+\infty} (g'(s+\lambda v))^2 \begin{bmatrix} 1 \\ \mathbf{u} \\ v \end{bmatrix} \begin{bmatrix} 1 & \mathbf{u}^{\mathbf{T}} & v \end{bmatrix} dv$$

$$= \begin{bmatrix} \rho_1 & \rho_1 \mathbf{u}^{\mathbf{T}} & \rho_2 \\ \rho_1 \mathbf{u} & \rho_1 \mathbf{u}\mathbf{u}^{\mathbf{T}} & \rho_2 \mathbf{u} \\ \rho_2 & \rho_2 \mathbf{u}^{\mathbf{T}} & \rho_3 \end{bmatrix} \tag{128}$$

where

$$\rho_1 \triangleq \int_{-\infty}^{+\infty} (g'(s+\lambda v))^2 dv = \int_{-\infty}^{+\infty} \left(\frac{2e^{-(s+\lambda v)}}{\left(1+e^{-(s+\lambda v)}\right)^2} \right)^2 dv \tag{129}$$

$$\rho_2 \triangleq \int_{-\infty}^{+\infty} (g'(s+\lambda v))^2 v dv = \int_{-\infty}^{+\infty} \left(\frac{2e^{-(s+\lambda v)}}{\left(1+e^{-(s+\lambda v)}\right)^2} \right)^2 v dv \tag{130}$$

$$\rho_3 \triangleq \int_{-\infty}^{+\infty} (g'(s+\lambda v))^2 v^2 dv = \int_{-\infty}^{+\infty} \left(\frac{2e^{-(s+\lambda v)}}{\left(1+e^{-(s+\lambda v)}\right)^2} \right)^2 v^2 dv \tag{131}$$

Solving the integral (127) basically amounts to finding the above three integrals, which are found to be:

$$\rho_1 = \frac{2}{3|\lambda|}, \quad \rho_2 = -\frac{2s}{3\lambda^2}, \quad \rho_3 = \frac{2}{3|\lambda|^3}\left(\tau + s^2\right) \tag{132}$$

where $\tau = \dfrac{\pi^2 - 6}{3} \cong 1.29$. Then,

$$\tilde{\boldsymbol{\Theta}}(\boldsymbol{\theta}, \lambda) = \int_{\mathbf{u}\in\mathfrak{D}} \begin{bmatrix} \rho_1 & \rho_1\mathbf{u}^{\mathrm{T}} & \rho_2 \\ \rho_1\mathbf{u} & \rho_1\mathbf{u}\mathbf{u}^{\mathrm{T}} & \rho_2\mathbf{u} \\ \rho_2 & \rho_2\mathbf{u}^{\mathrm{T}} & \rho_3 \end{bmatrix} d\mathbf{u}$$

$$= \begin{bmatrix} \dfrac{2}{3|\lambda|}\displaystyle\int_{\mathbf{u}\in\mathfrak{D}} d\mathbf{u} & \dfrac{2}{3|\lambda|}\displaystyle\int_{\mathbf{u}\in\mathfrak{D}} \mathbf{u}^{\mathrm{T}} d\mathbf{u} & -\dfrac{2}{3\lambda^2}\displaystyle\int_{\mathbf{u}\in\mathfrak{D}} s d\mathbf{u} \\ \dfrac{2}{3|\lambda|}\displaystyle\int_{\mathbf{u}\in\mathfrak{D}} \mathbf{u} d\mathbf{u} & \dfrac{2}{3|\lambda|}\displaystyle\int_{\mathbf{u}\in\mathfrak{D}} \mathbf{u}\mathbf{u}^{\mathrm{T}} d\mathbf{u} & -\dfrac{2}{3\lambda^2}\displaystyle\int_{\mathbf{u}\in\mathfrak{D}} s\mathbf{u} d\mathbf{u} \\ -\dfrac{2}{3\lambda^2}\displaystyle\int_{\mathbf{u}\in\mathfrak{D}} s d\mathbf{u} & -\dfrac{2}{3\lambda^2}\displaystyle\int_{\mathbf{u}\in\mathfrak{D}} s\mathbf{u}^{\mathrm{T}} d\mathbf{u} & \dfrac{2}{3|\lambda|^3}\left(\displaystyle\int_{\mathbf{u}\in\mathfrak{D}} \left(\tau + s^2\right) d\mathbf{u}\right) \end{bmatrix} \tag{133}$$

It is straightforward to obtain the following integrals contained in (133):

$$\int_{\mathbf{u}\in\mathfrak{D}} d\mathbf{u} = \prod_{i=1}^{n} b_i, \qquad \int_{\mathbf{u}\in\mathfrak{D}} \mathbf{u} d\mathbf{u} = 0, \qquad \int_{\mathbf{u}\in\mathfrak{D}} s d\mathbf{u} = \theta_0 \prod_{i=1}^{n} b_i, \qquad \int_{\mathbf{u}\in\mathfrak{D}} s\mathbf{u} d\mathbf{u} = \frac{1}{12}\prod_{i=1}^{n} b_i \begin{bmatrix} \theta_1 b_1^2 \\ \vdots \\ \theta_n b_n^2 \end{bmatrix} \tag{134}$$

$$\int_{\mathbf{u}\in\mathfrak{D}} \mathbf{u}\mathbf{u}^{\mathrm{T}} d\mathbf{u} = \frac{1}{12}\prod_{i=1}^{n} b_i \begin{bmatrix} b_1^2 & & \mathrm{O} \\ & \ddots & \\ \mathrm{O} & & b_n^2 \end{bmatrix}, \qquad \int_{\mathbf{u}\in\mathfrak{D}} \left(\tau + s^2\right) d\mathbf{u} = \prod_{i=1}^{n} b_i \left(\tau + \theta_0^2 + \frac{1}{12}\sum_{i=1}^{n} \theta_i^2 b_i^2\right) \tag{135}$$

Then (133) becomes

$$\tilde{\boldsymbol{\Theta}}(\boldsymbol{\theta}, \lambda) = \frac{2}{3|\lambda|}\prod_{i=1}^{n} b_i \begin{bmatrix} 1 & 0 & \cdots & 0 & -\frac{\theta_0}{|\lambda|} \\ 0 & \frac{b_1^2}{12} & \cdots & 0 & -\frac{\theta_1 b_1^2}{12|\lambda|} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \frac{b_n^2}{12} & -\frac{\theta_n b_n^2}{12|\lambda|} \\ -\frac{\theta_0}{|\lambda|} & -\frac{\theta_1 b_1^2}{12|\lambda|} & \cdots & -\frac{\theta_n b_n^2}{12|\lambda|} & \frac{1}{\lambda^2}\left(\tau + \theta_0^2 + \sum_{i=1}^{n}\frac{\theta_i^2 b_i^2}{12}\right) \end{bmatrix} \tag{136}$$

It is easy to verify that the inverse of $\tilde{\boldsymbol{\Theta}}(\boldsymbol{\theta}, \lambda)$ is given by

$$\tilde{\boldsymbol{\Theta}}^{-1}(\boldsymbol{\theta}, \lambda) = \frac{3|\lambda|}{2\prod_{i=1}^{n} b_i} \begin{bmatrix} 1+\frac{\theta_0^2}{\tau} & \frac{\theta_0\theta_1}{\tau} & \frac{\theta_0\theta_2}{\tau} & \cdots & \frac{\theta_0\theta_n}{\tau} & \frac{|\lambda|\theta_0}{\tau} \\ \frac{\theta_0\theta_1}{\tau} & \frac{12}{b_1^2}+\frac{\theta_1^2}{\tau} & \frac{\theta_1\theta_2}{\tau} & \cdots & \frac{\theta_1\theta_n}{\tau} & \frac{|\lambda|\theta_1}{\tau} \\ \frac{\theta_0\theta_2}{\tau} & \frac{\theta_1\theta_2}{\tau} & \frac{12}{b_3^2}+\frac{\theta_2^2}{\tau} & \cdots & \frac{\theta_2\theta_n}{\tau} & \frac{|\lambda|\theta_2}{\tau} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{\theta_0\theta_n}{\tau} & \frac{\theta_1\theta_n}{\tau} & \frac{\theta_2\theta_n}{\tau} & \cdots & \frac{12}{b_n^2}+\frac{\theta_n^2}{\tau} & \frac{|\lambda|\theta_n}{\tau} \\ \frac{|\lambda|\theta_0}{\tau} & \frac{|\lambda|\theta_1}{\tau} & \frac{|\lambda|\theta_2}{\tau} & \cdots & \frac{|\lambda|\theta_n}{\tau} & \frac{\lambda^2}{\tau} \end{bmatrix} \tag{137}$$

We further have

$$\tilde{\boldsymbol{\Theta}}^{-1}(\boldsymbol{\theta}, \lambda)\tilde{\mathbf{p}} = \frac{3|\lambda|}{2\prod_{i=1}^{n} b_i} g'(\tilde{s}) \begin{bmatrix} 1+\frac{\theta_0}{\tau}s+\frac{|\lambda|}{\tau}\theta_0 v \\ \frac{\theta_1}{\tau}s+\frac{12}{b_1^2}u_1+\frac{|\lambda|}{\tau}\theta_1 v \\ \vdots \\ \frac{\theta_n}{\tau}s+\frac{12}{b_n^2}u_n+\frac{|\lambda|}{\tau}\theta_n v \\ \frac{|\lambda|}{\tau}s+\frac{\lambda^2}{\tau}v \end{bmatrix} \tag{138}$$

And $$\tilde{\mathbf{p}}^{\mathbf{T}}\tilde{\mathbf{\Theta}}^{-1}(\boldsymbol{\theta},\lambda)\tilde{\mathbf{p}}=\frac{3|\lambda|}{2\prod_{i=1}^{n}b_i}g'^{2}(\tilde{s})\left(1+12\sum_{i=1}^{n}\frac{u_i^2}{b_i^2}+\frac{s^2+2s|\lambda|v+\lambda^2v^2}{\tau}\right) \tag{139}$$

Thus,

$$\begin{aligned}\delta\tilde{\boldsymbol{\theta}}^{opt}&=-\mu\frac{\tilde{\mathbf{p}}^{\mathbf{T}}\tilde{\mathbf{p}}}{\tilde{\mathbf{p}}^{\mathbf{T}}\tilde{\mathbf{\Theta}}^{-1}\tilde{\mathbf{p}}}\tilde{\mathbf{\Theta}}^{-1}\tilde{\mathbf{p}}\left(\mathbf{q}^{\mathbf{T}}\Delta\mathbf{y}\right)\\&=-\mu\frac{g'(\tilde{s})\left(1+\sum_{i=1}^{n}u_i^2+v^2\right)}{\left(1+12\sum_{i=1}^{n}\frac{u_i^2}{b_i^2}+\frac{s^2+2s|\lambda|v+\lambda^2v^2}{\tau}\right)}\begin{bmatrix}\frac{\theta_0 s}{\tau}+1+\frac{|\lambda|}{\tau}\theta_0 v\\ \frac{\theta_1 s}{\tau}+\frac{12}{b_1^2}u_1+\frac{|\lambda|}{\tau}\theta_1 v\\ \vdots\\ \frac{\theta_n s}{\tau}+\frac{12}{b_n^2}u_n+\frac{|\lambda|}{\tau}\theta_n v\\ \frac{|\lambda|}{\tau}s+\frac{\lambda^2}{\tau}v\end{bmatrix}\left(\mathbf{q}^{\mathbf{T}}\Delta\mathbf{y}\right)\end{aligned} \tag{140}$$

Letting $v=0$ in (140) yields (90).

Appendix B: C code of the PIL, BP and SDLM algorithms of MATLAB/Simulink s-function.

```
/*  S-Function for Multi-Layer Perceptron Neural Network
/*                  --- with standard BP learning algorithm and PIL algorithm for multi-input/multi-output networks
/*                  --- with one hidden layer
/*  Used under:    MATLAB 5.6/Simulink 5.0
/*  Date:          06/05/04
/*  Author:        Sheng Wan, PhD candidate, West Virginia University, Morganton, WV
/*  Version:       1.0
/*  Compiler:      Microsoft Visual C++ 6.0
/*  Modified:      02/10/05, to include the Stochastic Diagonal Levenberg-Marquardt (SDLM) method. Only valid
/*                  for single output with linear output neuron for this version.

/*
Multi-Layer Perceptron Neural Network:
(1)  n(=dim_in) input nodes; L(=NumOfNeurons) hidden neurons (1 hidden layer); m(=dim_out) output nodes.
(2)  Neuron activation function (for hidden and/or output layer):
                    Hyper tangent::     yhi = (1-exp(-Si))/(1+exp(-Si)),          i=1, 2, ..., L
                or          Logistic:   yhi = 1/(1+exp(-Si)),                     i=1, 2, ..., L
                                        Si = whi0 + whi1*u1 + ... + whin*un,      i=1, 2, ..., L
(3)  Output nodes can either be an identity function (linear), or the same activation function as that for hidden layer
(nonlinear).

Discrete time state variables:
  (a) input weights for hidden layer:
        x: 0                            to   dim_in,
                                                input weights for the 1st hidden neuron:  wh10, wh11, ..., wh1n
        x: dim_in+1                     to   2*dim_in+1,
                                                input weights for the 2nd hidden neuron:  wh20, wh22, ..., wh2n
  ..........................................................................................
        x: (L-1)*(dim_in+1)             to   L*(dim_in+1)-1,
                                                input weights for the Nth hidden neuron:  whL0, whL1, ..., whLn

  (b) input weights for output layer:
        x: L*(dim_in+1)                 to   L*(dim_in+2),
                                                input weights for the 1st output neuron:  wy10, wy11, ..., wy1L
        x: L*(dim_in+2)+1               to   L*(dim_in+3)+1,
                                                input weights for the 2nd output neuron:  wy20, wy21, ..., wy2L
        ..........................................................
        ..........................................................
        x: L*(dim_in+dim_out)+dim_out-1   to   L*(dim_in+dim_out+1)+dim_out-1,
                                                input weights for the last output neuron: wym0, wym1, ..., wymL
  (c) running estimate of the diagonal second derivative (one-to-one correspondent with the weights)
      (only for the stochastic diagonal Levenberg-Marquardt method):
        x: L*(dim_in+dim_out+1)+dim_out   to   2*( L*(dim_in+dim_out+1)+dim_out )-1

  (d) the last state is a counter used to control the states log (output) cycle:
        x: 2*( L*(dim_in+dim_out+1)+dim_out )

WORK variables:
        (a)  for hidden neurons:
                0                           to   NumOfNeurons-1                 hidden neuron's input
                NumOfNeurons                to   2*NumOfNeurons-1               hidden neuron's output
        (b)  for output neurons:
                2*NumOfNeurons              to   2*NumOfNeurons+dim_out-1       output neuron's input
                2*NumOfNeurons+dim_out      to   2*(NumOfNeurons+dim_out)-1     output neuron's output
        (c)  for error backpropagation:
```

2*(NumOfNeurons+dim_out) to 2*(NumOfNeurons+dim_out)+dim_out-1

Parameters are presented in the following order and are all scalar except the last one:

0. Input/output dimension
1. Maximum number of neurons
2. Sampling period
3. Learning method (Sequential = 1, Gradient descent = 2, Other = 3, SDLM = 4)
4. Learning rate
5. Activation function: sinh/sigmoid
6. Output neuron type (linear/non-linear)
7. Initial neural network parameter vector
8. mu - a small number used by the SDLM method to prevent the algorithm from blowing up
9. lamda - filter constant used by the SDLM method
10. Cycle number used to control the log (output) of the weights (states)

*/

```
#define S_FUNCTION_NAME mlpnn_plus
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"
#include <math.h>

#define MDL_CHECK_PARAMETERS
static void mdlCheckParameters(SimStruct *S) {}

/* mdlInitializeSizes - initialize the sizes array ********************************************/
static void mdlInitializeSizes(SimStruct *S)
{
        real_T *ptr;
        int_T  dim_in, dim_out, NumOfNeurons;

        ptr=mxGetPr(ssGetSFcnParam(S,0));    dim_in=(int_T)ptr[0]; dim_out=(int_T)ptr[1];
        ptr=mxGetPr(ssGetSFcnParam(S,1));    NumOfNeurons=(int_T)ptr[0];

   ssSetNumSFcnParams(S,8+3);                                   /* number of expected parameters       */
   if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S))
        { mdlCheckParameters(S); if (ssGetErrorStatus(S) != NULL) return; } else return;
        ssSetNumContStates(S,0);                                /* number of continuous states         */
   ssSetNumDiscStates(S,2*(NumOfNeurons*(dim_in+dim_out+1)+dim_out)+1);
                                                                /* number of discrete states    */
   if (!ssSetNumInputPorts(S,3)) return;                         /* number of input ports               */
   ssSetInputPortWidth(S,0,dim_in);                             /* first input port width (# of inputs) */
        ssSetInputPortDirectFeedThrough(S,0,1);                 /* first port direct feedthrough flag   */
   ssSetInputPortWidth(S,1,dim_out);                            /* second input port width (error)      */
        ssSetInputPortDirectFeedThrough(S,1,0);                 /* second port direct feedthrough flag  */
   ssSetInputPortWidth(S,2,1);                                  /* third input port width (Enable)      */
   ssSetInputPortDirectFeedThrough(S,2,0);                      /* third port direct feedthrough flag   */
        if (!ssSetNumOutputPorts(S,2)) return;                  /* number of output ports              */
        ssSetOutputPortWidth(S,0,dim_out);                      /* first output port width             */
        ssSetOutputPortWidth(S,1,2*(NumOfNeurons*(dim_in+dim_out+1)+dim_out)+1);
                                                                /* second output port width            */
        ssSetNumSampleTimes(S,1);                               /* number of sample times              */
        ssSetNumRWork(S,2*NumOfNeurons+3*dim_out);       /* number real_T work vector elements   */
        ssSetNumIWork(S,0);                                     /* number int_T work vector elements    */
        ssSetNumPWork(S,0);                                     /* number ptr work vector elements      */
        ssSetNumModes(S,0);                                     /* number mode work vector elements     */
        ssSetNumNonsampledZCs(S,0);                             /* number of nonsampled zero crossing   */
```

```
}

/* mdlInitializeSampleTimes - initialize the sample times array ******************************/
static void mdlInitializeSampleTimes(SimStruct *S)
{
	real_T  *T=mxGetPr(ssGetSFcnParam(S,2));
	ssSetSampleTime(S, 0, T[0]);
	ssSetOffsetTime(S, 0, 0);
}

/* mdlStart - initialize work vectors ********************************************************/
#define MDL_START
static void mdlStart(SimStruct *S)
{
	real_T  *ptr, *WORK;
	int_T NumOfNeurons, dim_out, i;

	ptr=mxGetPr(ssGetSFcnParam(S,1));	NumOfNeurons=(int_T)ptr[0];
	ptr=mxGetPr(ssGetSFcnParam(S,0));	dim_out=(int_T)ptr[1];

	WORK=ssGetRWork(S);
	for(i=0; i<2*NumOfNeurons+3*dim_out; i++) WORK[i]=0.0;
}

/* mdlInitializeConditions - initialize the states *******************************************/
#define MDL_INITIALIZE_CONDITIONS
static void mdlInitializeConditions(SimStruct *S)
{
	real_T  *x0  = ssGetRealDiscStates(S);
	real_T  *pr, *px0;
	int_T dim_in, dim_out, NumOfNeurons, i;

	pr=mxGetPr(ssGetSFcnParam(S,0));	dim_in=(int_T)pr[0]; dim_out=(int_T)pr[1];
	pr=mxGetPr(ssGetSFcnParam(S,1));	NumOfNeurons=(int_T)pr[0];
	px0=mxGetPr(ssGetSFcnParam(S,7));

	for(i=0;i<2*(NumOfNeurons*(dim_in+dim_out+1)+dim_out)+1;i++)  x0[i]=px0[i];
}

static void mdlOutputs(SimStruct *S, int_T tid)
{
	real_T   *x     = ssGetRealDiscStates(S);
	real_T   *y     = ssGetOutputPortRealSignal(S,0);
	real_T   *weight = ssGetOutputPortRealSignal(S,1);

	InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);

	real_T  *pr, *WORK;
	int_T dim_in, dim_out, NumOfNeurons, FuncType, OutputType, i, j;

	pr=mxGetPr(ssGetSFcnParam(S,0));	dim_in     =(int_T)pr[0];
	dim_out    =(int_T)pr[1];
	pr=mxGetPr(ssGetSFcnParam(S,1));	NumOfNeurons=(int_T)pr[0];
	pr=mxGetPr(ssGetSFcnParam(S,5));	FuncType   =(int_T)pr[0];
	pr=mxGetPr(ssGetSFcnParam(S,6));	OutputType  =(int_T)pr[0];

	WORK=ssGetRWork(S);
```

```
	for (i=0;i<NumOfNeurons;i++)    // hidden layer
	{
		WORK[i]=x[i*(dim_in+1)];
		for (j=0;j<dim_in;j++) WORK[i] += x[i*(dim_in+1)+1+j]*(*uPtrs[j]);                    // hidden neuron's input

		if (FuncType == 1) // tanh functoin
			WORK[NumOfNeurons+i]=(1.0-exp(-WORK[i]))/(1.0+exp(-WORK[i])); // hidden neuron's output
		else                    // logistic functoin
			WORK[NumOfNeurons+i]=1.0/(1.0+exp(-WORK[i]));                    // hidden neuron's output

	}

	// output layer neuron's input
	for (i=0;i<dim_out;i++)       WORK[2*NumOfNeurons+i] = x[NumOfNeurons*(dim_in+1+i)+i];
	for (i=0;i<dim_out;i++)
	{
		for (j=0; j<NumOfNeurons; j++)
			WORK[2*NumOfNeurons+i] +=
					x[NumOfNeurons*(dim_in+1+i)+i+j+1]*WORK[NumOfNeurons+j];
	}

	// output layer neuron's output
	if (OutputType == 1)                  // nonlinear output neuron
	{
		if (FuncType == 1) // tanh functoin
		{
			for (i=0;i<dim_out;i++)
			{
				WORK[2*NumOfNeurons+dim_out+i]
				= (1.0-exp(-WORK[2*NumOfNeurons+i]))/(1.0+exp(-WORK[2*NumOfNeurons+i]));
			}
		}
		else                    // logistic functoin
		{
			for (i=0;i<dim_out;i++)
				WORK[2*NumOfNeurons+dim_out+i] = 1.0/(1.0+exp(-WORK[2*NumOfNeurons+i]));
		}
		for (i=0;i<dim_out;i++) y[i] = WORK[2*NumOfNeurons+dim_out+i]; // network output
	}
	else                            // linear output neuron
	{
		for (i=0;i<dim_out;i++)
		{
			WORK[2*NumOfNeurons+dim_out+i]=WORK[2*NumOfNeurons+i];
			y[i] = WORK[2*NumOfNeurons+i];
		}
	}

	for(i=0;i<2*((dim_in+dim_out+1)*NumOfNeurons+dim_out)+1;i++)
		weight[i]=x[i];                         // log neural network states (weights)

}

#define MDL_UPDATE
static void mdlUpdate(SimStruct *S, int_T tid)
{
```

```
real_T  *pr, eta, *WORK;
real_T  *x= ssGetRealDiscStates(S);
real_T  temp, temp1, temp2, SumOfx2, SumOfy2,SumOfw2;
int_T   dim_in, dim_out, NumOfNeurons, i, j, Learning_Method, FuncType, OutputType, Of2nd;
real_T  tao, bao, k, c, d, eps, mu, lamda;

InputRealPtrsType LE = ssGetInputPortRealSignalPtrs(S,2);

if ( (*LE[0]!=0) )
{
        InputRealPtrsType u  = ssGetInputPortRealSignalPtrs(S,0);
        InputRealPtrsType e  = ssGetInputPortRealSignalPtrs(S,1);
        pr=mxGetPr(ssGetSFcnParam(S,0));     dim_in=(int_T)pr[0];    dim_out=(int_T)pr[1];
        pr=mxGetPr(ssGetSFcnParam(S,1));     NumOfNeurons=(int_T)pr[0];
        pr=mxGetPr(ssGetSFcnParam(S,3));     Learning_Method=(int_T)pr[0];
        pr=mxGetPr(ssGetSFcnParam(S,4));     eta=pr[0];
        pr=mxGetPr(ssGetSFcnParam(S,5));     FuncType    =(int_T)pr[0];
        pr=mxGetPr(ssGetSFcnParam(S,6));     OutputType  =(int_T)pr[0];
        pr=mxGetPr(ssGetSFcnParam(S,8));     mu          =pr[0];
        pr=mxGetPr(ssGetSFcnParam(S,9));     lamda  =pr[0];

        WORK=ssGetRWork(S);

        switch (Learning_Method-1)
        {
        case 0:  // Standard PIL method
                c   = (3.1416*3.1416-6.0)/3.0;
                bao = 3.0;
                // output neuron's weights update
                SumOfy2 = 0.0;          // square sum of the hidden neuron's output
                for (i=0;i<NumOfNeurons;i++)
                        SumOfy2 += WORK[NumOfNeurons+i]*WORK[NumOfNeurons+i];

                if (OutputType == 1)    // nonlinear output neuron
                {
                        for (i=0;i<dim_out;i++)  // for i th output neuron
                        {

                                tao = 1/c;

                                // prepare for error back-propagation

                                if (FuncType == 1)    // tanh function
                                        WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i])*
                        (1.0-WORK[2*NumOfNeurons+dim_out+i]*WORK[2*NumOfNeurons+dim_out+i])/2.0;
                                else                  // logistic function
                                        WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i])*
                        WORK[2*NumOfNeurons+dim_out+i]*(1-WORK[2*NumOfNeurons+dim_out+i]);

                                temp = WORK[2*(NumOfNeurons+dim_out)+i]*(1.0+SumOfy2)/
                        (1.0+WORK[2*NumOfNeurons+i]*WORK[2*NumOfNeurons+i]*tao+bao*SumOfy2);

                                x[NumOfNeurons*(dim_in+1+i)+i]         // output bias adaptation
                        +=temp*(1.0+x[NumOfNeurons*(dim_in+1+i)+i]*WORK[2*NumOfNeurons+i]*tao);

                                for (j=0;j<NumOfNeurons;j++)             // output weights adaptation
                                {
```

```
                x[NumOfNeurons*(dim_in+1+i)+i+1+j]+=temp*(x[NumOfNeurons*(dim_in+1+i)+i+1+j]
                        *WORK[2*NumOfNeurons+i]*tao + bao*WORK[NumOfNeurons+j]);
                }
            }
        }
        else            // linear output neuron
        {
            for (i=0;i<dim_out;i++)  // prepare for error back-propagation
                WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i]);

            temp = eta*(1.0+SumOfy2)/(1.0+bao*SumOfy2);
            for (i=0;i<dim_out;i++)  // for i th output neuron
            {
                x[NumOfNeurons*(dim_in+1+i)+i] += temp*(*e[i]);  // output bias adaptation
                for (j=0;j<NumOfNeurons;j++)                    // output weights adaptation
            x[NumOfNeurons*(dim_in+1+i)+1+i+j]+=temp*(*e[i])*bao*WORK[NumOfNeurons+j];
            }
        }
        // hidden neuron's weights adaptation
        SumOfx2 = 0.0;
        for (j=0;j<dim_in;j++)
        {
            SumOfx2 += (*u[j])*(*u[j]);
        }
        for (i=0;i<NumOfNeurons;i++)  // for i th hidden neuron
        {
            tao = 1/c;
            temp = 0.0;
            for (j=0;j<dim_out;j++)
            temp+=WORK[2*(NumOfNeurons+dim_out)+j]*x[NumOfNeurons*(dim_in+1+j)+1+i+j];

            if (FuncType == 1) // tanh functoin
                temp *= (1.0-WORK[NumOfNeurons+i]*WORK[NumOfNeurons+i])/2.0;
            else                // logistic functoin
                temp *= (1.0-WORK[NumOfNeurons+i])*WORK[NumOfNeurons+i];

            temp *= (1.0+SumOfx2)/(1.0+WORK[i]*WORK[i]*tao+bao*SumOfx2);
            x[i*(dim_in+1)] += temp*(1.0+x[i*(dim_in+1)]*WORK[i]*tao); // input bias adaptation
            for (j=0;j<dim_in;j++)                                      // input weights adaptation
            {
                x[i*(dim_in+1)+1+j] += temp*( x[i*(dim_in+1)+1+j]*WORK[i]*tao+bao*(*u[j]) );
            }
        }
        break;
case 1:  // Gradient descent
        // output neuron's weights update
        if (OutputType == 1)    // nonlinear output neuron
        {
            for (i=0;i<dim_out;i++)  // for i th output neuron
            {                        // prepare for error back-propagation
                if (FuncType == 1)    // tanh function
                    WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i])*
            (1.0-WORK[2*NumOfNeurons+dim_out+i]*WORK[2*NumOfNeurons+dim_out+i])/2.0;
                else                    // logistic function
                    WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i])*
            WORK[2*NumOfNeurons+dim_out+i]*(1-WORK[2*NumOfNeurons+dim_out+i]);
                x[NumOfNeurons*(dim_in+1+i)+i]        // output bias adaptation
```

```
                                        += WORK[2*(NumOfNeurons+dim_out)+i];

                                for (j=0;j<NumOfNeurons;j++)          // output weights adaptation
                                {
                                        x[NumOfNeurons*(dim_in+1+i)+i+1+j] +=
                                        WORK[2*(NumOfNeurons+dim_out)+i]*WORK[NumOfNeurons+j];
                                }
                        }
                }
                else                    // linear output neuron
                {
                        for (i=0;i<dim_out;i++)  // prepare for error back-propagation
                                WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i]);

                        for (i=0;i<dim_out;i++)  // for i th output neuron
                        {
                                x[NumOfNeurons*(dim_in+1+i)+i] += eta*(*e[i]);  // output bias adaptation
                                for (j=0;j<NumOfNeurons;j++)                    // output weights adaptation
                                        x[NumOfNeurons*(dim_in+1+i)+1+i+j] +=
                                        eta*(*e[i])*WORK[NumOfNeurons+j];
                        }
                }
                // hidden neuron's weights adaptation
                for (i=0;i<NumOfNeurons;i++)  // for i th hidden neuron
                {
                        temp = 0.0;
                        for (j=0;j<dim_out;j++)
                        temp+=WORK[2*(NumOfNeurons+dim_out)+j]*x[NumOfNeurons*(dim_in+1+j)+1+i+j];

                        if (FuncType == 1) // tanh functoin
                                temp *= (1.0-WORK[NumOfNeurons+i]*WORK[NumOfNeurons+i])/2.0;
                        else                    // logistic functoin
                                temp *= (1.0-WORK[NumOfNeurons+i])*WORK[NumOfNeurons+i];
                        x[i*(dim_in+1)] += temp;                        // input bias adaptation
                        for (j=0;j<dim_in;j++)                          // input weights adaptation
                        {
                                x[i*(dim_in+1)+1+j] += temp*(*u[j]);
                        }
                }
                break;
case 2:  // Modified PIL method
                k   = 0.0;
                c   = 1;
                eps = 0.0;
                bao = 1.0; d=0.3+0.2+0.1;
                // output neuron's weights update
                SumOfy2 = 0.0;          // square sum of the hidden neuron's output
                for (i=0;i<NumOfNeurons;i++)
                        SumOfy2 += WORK[NumOfNeurons+i]*WORK[NumOfNeurons+i];

                if (OutputType == 1)    // nonlinear output neuron
                {
                        for (i=0;i<dim_out;i++)  // for i th output neuron
                        {
                        SumOfw2=x[NumOfNeurons*(dim_in+1+i)+i]*x[NumOfNeurons*(dim_in+1+i)+i];
                                for (j=0;j<NumOfNeurons;j++)
                SumOfw2+=x[NumOfNeurons*(dim_in+1+i)+i+1+j]*x[NumOfNeurons*(dim_in+1+i)+i+1+j];
```

```
                                        SumOfw2 = SumOfw2/(NumOfNeurons+1.0);
                                        tao = k/(c + eps*SumOfw2);

                                        // prepare for error back-propagation

                                        if (FuncType == 1)    // tanh functoin
                                                WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i])*
                                (1.0-WORK[2*NumOfNeurons+dim_out+i]*WORK[2*NumOfNeurons+dim_out+i])/2.0;
                                        else                    // logistic functoin
                                                WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i])*
                                WORK[2*NumOfNeurons+dim_out+i]*(1-WORK[2*NumOfNeurons+dim_out+i]);

                                        temp = WORK[2*(NumOfNeurons+dim_out)+i]*(1.0+SumOfy2)/
                                (1.0+WORK[2*NumOfNeurons+i]*WORK[2*NumOfNeurons+i]*tao+bao*SumOfy2);

                                        x[NumOfNeurons*(dim_in+1+i)+i]        // output bias adaptation
                                +=temp*(1.0+x[NumOfNeurons*(dim_in+1+i)+i]*WORK[2*NumOfNeurons+i]*(tao+d));

                                        for (j=0;j<NumOfNeurons;j++)            // output weights adaptation
                                        {
                                                x[NumOfNeurons*(dim_in+1+i)+i+1+j]
+=temp*(x[NumOfNeurons*(dim_in+1+i)+i+1+j]*WORK[2*NumOfNeurons+i]*(tao+d) + bao*WORK[NumOfNeurons+j]);
                                        }
                                }
                        }
                        else                // linear output neuron
                        {
                                for (i=0;i<dim_out;i++)  // prepare for error back-propagation
                                        WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i]);

                                temp = eta*(1.0+SumOfy2)/(1.0+bao*SumOfy2);
                                for (i=0;i<dim_out;i++)  // for i th output neuron
                                {
                                        x[NumOfNeurons*(dim_in+1+i)+i] += temp*(*e[i]);  // output bias adaptation
                                        for (j=0;j<NumOfNeurons;j++)                    // output weights adaptation
                                x[NumOfNeurons*(dim_in+1+i)+1+i+j]+=temp*(*e[i])*bao*WORK[NumOfNeurons+j];
                                }
                        }

                        // hidden neuron's weights adaptation
                        SumOfx2 = 0.0;
                        for (j=0;j<dim_in;j++)
                        {
                                SumOfx2 += (*u[j])*(*u[j]);
                        }
                        for (i=0;i<NumOfNeurons;i++)  // for i th hidden neuron
                        {
                                SumOfw2 = x[i*(dim_in+1)]*x[i*(dim_in+1)];
                                for (j=0;j<dim_in;j++)
                                        SumOfw2 += x[i*(dim_in+1)+1+j]*x[i*(dim_in+1)+1+j];
                                SumOfw2 = SumOfw2/(dim_in+1.0);
                                tao = k/(c + eps*SumOfw2);

                                temp = 0.0;
                                for (j=0;j<dim_out;j++)
                                temp+=WORK[2*(NumOfNeurons+dim_out)+j]*x[NumOfNeurons*(dim_in+1+j)+1+i+j];
```

```
                        if (FuncType == 1) // tanh function
                                temp *= (1.0-WORK[NumOfNeurons+i]*WORK[NumOfNeurons+i])/2.0;
                        else                    // logistic function
                                temp *= (1.0-WORK[NumOfNeurons+i])*WORK[NumOfNeurons+i];

                        temp *= (1.0+SumOfx2)/(1.0+WORK[i]*WORK[i]*tao+bao*SumOfx2);
                        x[i*(dim_in+1)] += temp*(1.0+x[i*(dim_in+1)]*WORK[i]*(tao+d));
                                                                // input bias adaptation
                        for (j=0;j<dim_in;j++)                  // input weights adaptation
                        {
                        x[i*(dim_in+1)+1+j]+=temp*( x[i*(dim_in+1)+1+j]*WORK[i]*(tao+d)+bao*(*u[j]) );
                        }
                }
                break;
        case 3:  // Stochastic Diagonal Levenberg-Marquardt (SDLM) method, only for 1 output.
                Of2nd = NumOfNeurons*(dim_in+dim_out+1)+dim_out;
                // output neuron's weights update
                if (OutputType == 1)    // nonlinear output neuron
                {
                        for (i=0;i<dim_out;i++)  // for i th output neuron, (only 1 output considered => i = 0)
                        {
                                // prepare for error back-propagation

                                if (FuncType == 1)    // tanh function
                                {
                temp2 =(1.0-WORK[2*NumOfNeurons+dim_out+i]*WORK[2*NumOfNeurons+dim_out+i])/2.0;
                                        WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i])*temp2;
                                }
                                else                    // logistic function
                                {
                temp2 = WORK[2*NumOfNeurons+dim_out+i]*(1-WORK[2*NumOfNeurons+dim_out+i]);
                                        WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i])*temp2;

                                }
                                x[Of2nd+NumOfNeurons*(dim_in+1+i)+i]
                                        = (1.0-lamda)*x[Of2nd+NumOfNeurons*(dim_in+1+i)+i]
                                          + lamda*temp2*temp2;
                                                // running estimate of the diagonal second derivative
                                x[NumOfNeurons*(dim_in+1+i)+i]          // output bias adaptation
                += WORK[2*(NumOfNeurons+dim_out)+i]/(mu+x[Of2nd+NumOfNeurons*(dim_in+1+i)+i]);

                                for (j=0;j<NumOfNeurons;j++)            // output weights adaptation
                                {
                                temp = (temp2*WORK[NumOfNeurons+j])*(temp2*WORK[NumOfNeurons+j]);
x[Of2nd+NumOfNeurons*(dim_in+1+i)+i+1+j]= (1.0-lamda)*x[Of2nd+NumOfNeurons*(dim_in+1+i)+i+1+j]+lamda*temp;
                                        temp = mu+x[Of2nd+NumOfNeurons*(dim_in+1+i)+i+1+j];
x[NumOfNeurons*(dim_in+1+i)+i+1+j] += WORK[2*(NumOfNeurons+dim_out)+i]*WORK[NumOfNeurons+j]/temp;
                                }
                        }
                }
                else                    // linear output neuron
                {
                        for (i=0;i<dim_out;i++)  // prepare for error back-propagation
                                WORK[2*(NumOfNeurons+dim_out)+i] = eta*(*e[i]);

                        for (i=0;i<dim_out;i++)  // for i th output neuron
                        {
```

```
                                        x[NumOfNeurons*(dim_in+1+i)+i] += eta*(*e[i]);  // output bias adaptation
                                        for (j=0;j<NumOfNeurons;j++)                    // output weights adaptation
                                        {
                                                x[Of2nd+NumOfNeurons*(dim_in+1+i)+1+i+j]
=(1.0-lamda)*x[Of2nd+NumOfNeurons*(dim_in+1+i)+1+i+j]+lamda*WORK[NumOfNeurons+j]*WORK[NumOfNeurons+j];
                                                temp = mu+x[Of2nd+NumOfNeurons*(dim_in+1+i)+1+i+j];
                                        x[NumOfNeurons*(dim_in+1+i)+1+i+j] += eta*(*e[i])*WORK[NumOfNeurons+j]/temp;
                                        }
                                }
                        }
                        // hidden neuron's weights adaptation
                        for (i=0;i<NumOfNeurons;i++)  // for i th hidden neuron
                        {
                                temp = 0.0;
                                for (j=0;j<dim_out;j++)
                        temp += WORK[2*(NumOfNeurons+dim_out)+j]*x[NumOfNeurons*(dim_in+1+j)+1+i+j];

                                if (FuncType == 1) // tanh function
                                        temp1 = (1.0-WORK[NumOfNeurons+i]*WORK[NumOfNeurons+i])/2.0;
                                else                    // logistic function
                                        temp1 = (1.0-WORK[NumOfNeurons+i])*WORK[NumOfNeurons+i];
                                temp *= temp1;
                                temp1 *= x[NumOfNeurons*(dim_in+1)+1+i];
                                if (OutputType == 1)    // nonlinear output neuron
                                        temp1 *= temp2;
                                temp1 *= temp1;
                                x[Of2nd+i*(dim_in+1)] = (1.0-lamda)*x[Of2nd+i*(dim_in+1)]+lamda*temp1;
                                x[i*(dim_in+1)] += temp/(mu+x[Of2nd+i*(dim_in+1)]);   // input bias adaptation
                                for (j=0;j<dim_in;j++)                                // input weights adaptation
                                {
                        x[Of2nd+i*(dim_in+1)+1+j] = (1.0-lamda)*x[Of2nd+i*(dim_in+1)+1+j]+lamda*temp1*(*u[j])*(*u[j]);
                                        x[i*(dim_in+1)+1+j] += temp*(*u[j])/(mu+x[Of2nd+i*(dim_in+1)+1+j]);
                                }
                        }
                        break;
                }
        }
}

/* mdlTerminate - called when the simulation is terminated ***********************************/
static void mdlTerminate(SimStruct *S) {}

/* Trailer information to set everything up for simulink usage ******************************/
#ifdef  MATLAB_MEX_FILE         /* Is this file being compiled as a MEX-file?   */
#include "simulink.c"           /* MEX-file interface mechanism                 */
#else
#include "cg_sfun.h"            /* Code generation registration function        */
#endif

#undef DIM
```